How to Use this Book

This guide is intended for application developers creating programs using the OpenDoc interface for OS/2 Warp who want to gain an understanding of the OpenDoc programming environment and to develop part editors.

Before you begin to use this information, it would be helpful to understand how you can:

- Expand the Contents to see all available topics
- Obtain additional information for a highlighted word or phrase
- Use action bar choices
- Use the programming information.

How to Use the Contents

When the Contents window first appears, some topics have a plus (+) sign beside them. The plus sign indicates that additional topics are available.

To expand the Contents if you are using a mouse, click on the plus sign. If you are using the keyboard, use the Up or Down Arrow key to highlight the topic, and press the plus (+) key. For example, **Code Pages** has a plus sign beside it. To see additional topics for that heading, click on the plus sign or highlight that topic and press the plus (+) key.

To view a topic, double-click on the topic (or press the Up or Down Arrow key to highlight the topic, and then press the Enter key).

How to Obtain Additional Information

After you select a topic, the information for that topic appears in a window. Highlighted words or phrases indicate that additional information is available. You will notice that certain words and phrases are highlighted in green letters, or in white letters on a black background. These are called hypertext terms. If you are using a mouse, double-click on the highlighted word. If you are using a keyboard, press the Tab key to move to the highlighted word, and then press the Enter key. Additional information then appears in a window.

How to Use Action Bar Choices

Several choices are available for managing information presented in the *OpenDoc Class Reference for the Macintosh* . There are three pull-down menus on the action bar: the **Services** menu, the **Options** menu, and the **Help** menu.

The actions that are selectable from the **Services** menu operate on the active window currently displayed on the screen. These actions include the following:

Bookmark

Allows you to set a placeholder so you can retrieve information of interest to you.

When you place a bookmark on a topic, it is added to a list of bookmarks you have previously set. You can view the list, and you can remove one or all bookmarks from the list. If you have not set any bookmarks, the list is empty.

To set a bookmark, do the following:

1. Select a topic from the Contents.
2. When that topic appears, choose the **Bookmark** option from the **Services** pull-down.
3. If you want to change the name used for the bookmark, type the new name in the field.
4. Click on the **Place** radio button (or press the Up or Down Arrow key to select it).
5. Click on **OK** (or select it and press Enter). The bookmark is then added to the bookmark list.

Search

Allows you to find occurrences of a word or phrase in the current topic, selected topics, or all topics.

You can specify a word or phrase to be searched. You can also limit the search to a set of topics by first marking the topics in the Contents list.

To search for a word or phrase in all topics, do the following:

1. Choose the **Search** option from the **Services** pull-down.
2. Type the word or words to be searched for.
3. Click on **All sections** (or press the Up or Down Arrow keys to select it).
4. Click on **Search** (or select it and press Enter) to begin the search.
5. The list of topics where the word or phrase appears is displayed.

Print

Allows you to print one or more topics. You can also print a set of topics by first marking the topics in the Contents list.

To print the document Contents list, do the following:

1. Choose **Print** from the **Services** pull-down.
2. Click on **Contents** (or press the Up or Down Arrow key to select it).
3. Click on **Print** (or select it and press Enter).
4. The Contents list is printed on your printer.

Copy

Allows you to copy a topic that you are viewing to the System Clipboard or to a file that you can edit. You will find this particularly useful for copying syntax definitions and program samples into the application that you are developing.

You can copy a topic that you are viewing in two ways:

- **Copy** copies the topic that you are viewing into the System Clipboard. If you are using a Presentation Manager editor (for example, the System Editor) that copies or cuts (or both) to the System Clipboard, and pastes to the System Clipboard, you can easily add the copied information to your program source module.
- **Copy to file** copies the topic that you are viewing into a temporary file named TEXT.TMP. You can later edit that file by using any editor. You will find TEXT.TMP in the directory where your viewable document resides.

  To copy a topic, do the following:

  1. Expand the Contents list and select a topic.
  2. When the topic appears, choose **Copy to file** from the **Services** pull-down.
  3. The system puts the text pertaining to that topic into the temporary file named TEXT.TMP.

For information on one of the other choices in the **Services** pull-down, highlight the choice and press the F1 key.

The actions that are selectable from the **Options** menu allow you to change the way your Contents list is displayed. To expand the Contents and show all levels for all topics, choose **Expand all** from the **Options** pull-down. You can also press the Ctrl and * keys together. For information on one of the other choices in the **Options** pull-down, highlight the choice and press the F1 key.

The actions that are selectable from the **Help** menu allow you to select different types of help information. You can also press the F1 key for help information about the Information Presentation Facility (IPF).

OpenDoc Fundamentals

OpenDoc is an architecture designed to enable the construction of compound, collaborative, customizable, and cross-platform applications. It enables the creation of compound documents, which are created and edited by several cooperating applications working within a single document. OpenDoc also supports scripting and extension mechanisms that allow for communication among parts of a document. OpenDoc allows for multiple parts in a single document. These different parts can range from a spread sheet to voice inside of the document.

OpenDoc consists of a set of shared libraries that can be used to build editors and viewers for compound documents, as well as other compound software to provide services to documents.

About Component Integration Laboratories

OpenDoc is presented and maintained through a nonprofit organization devoted to promoting cross-platform standards, architectures, and protocols in a vendor-independent fashion. This organization, Component Integration Laboratories (CI Labs), is composed of a number of platform and application vendors with a common interest in solving OpenDoc issues and promoting interoperability.

CI Labs supports several levels of participation through different membership categories. If you are interested in shaping the future direction of component software, or if you simply need to be kept abreast of the latest developments, you can become a member. For an information packet, send your mailing address to:

Component Integration Laboratories

PO Box 61747
Sunnyvale, CA 94088-1747

Telephone: 408-864-0300
FAX: 408-864-0380
Internet: cilabs@cilabs.org
World Wide Web: http://www.cilabs.org

Introduction

OpenDoc is a revolutionary technology that brings a new class of applications and documents to the Windows, Mac OS, OS/2, UNIX, and other personal-computer platforms. With OpenDoc, hardware and software developers can deliver:

- New software technologies to individual users
- Better server integration to corporate users
- Enhanced multimedia content to all users

OpenDoc enables the creation of a new kind of software. This cooperative component software:

- Supports compound documents
- Can be customized
- Can be used collaboratively across networks
- Is available across multiple platforms

In doing so, OpenDoc fundamentally changes the nature of software development for personal computers.

This chapter starts with a review of the reasons why OpenDoc was created, followed by an overview of OpenDoc concepts:

- How OpenDoc documents are structured
- How OpenDoc software handles events and user interaction
- How to extend OpenDoc's capabilities
- How to ensure cross-platform compatibility for your OpenDoc software

Each of these topics is described more fully in subsequent chapters.

Why OpenDoc?

Customer demand for increasingly sophisticated and integrated software solutions has led to large and sometimes unwieldy application packages, feature-laden but difficult to maintain and modify. Developing, maintaining, and upgrading these large, cumbersome applications can require a vast organization. The programs are difficult to create, expensive to maintain, and can take years to revise.

Upgrades or bug fixes in one component of such an application require the developer to release-and the customer to buy-a completely new version of the entire package. Some developers have added extension mechanisms to their packages to allow addition or replacement of certain components, but the extensions are proprietary, incompatible with other applications, and applicable to only certain parts of the package.

Because of the barriers put up by the current application architecture, users are often frustrated in attempting to perform common tasks:

- Users often cannot assemble complex documents from multiple sources because of the many error-prone, manual operations required.

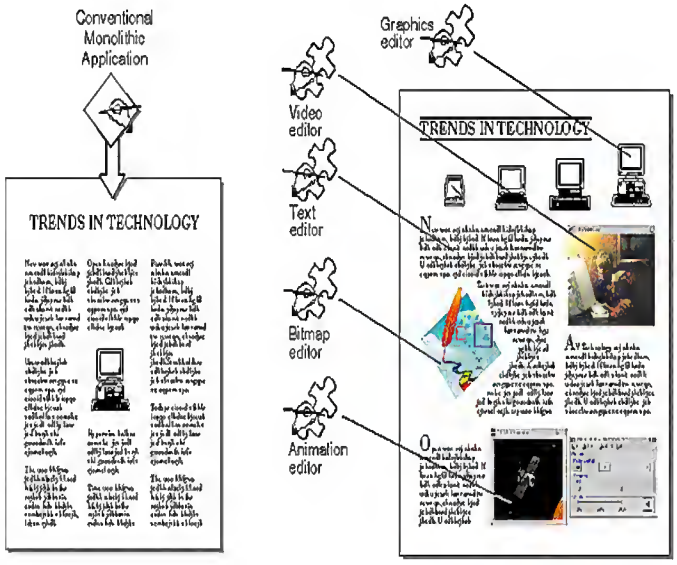
- Users often cannot edit different kinds of content within a single document. Most applications support only one or a few different kinds of content, such as a single format for text and a single format for graphics. Furthermore, the editing procedures for a given kind of content are different across applications, complicating data transfer among applications.

- Business users are forced to choose between the reliability of shrink-wrapped software and the extra features of custom solutions designed for their needs. To increase reliability while maintaining and adding custom features, businesses need simple and standardized designs and procedures.

- Users may not be able to use an editor or tool provided by an individual developer or small development team because the editor may be incompatible with the users' current application package, and the developer may not have sufficient resources to develop an entire integrated package.

- Computers often frustrate users' efforts to collaborate with others. Users cannot share documents across applications, recover changes to shared documents, or, with rare exceptions, manipulate the contents of one document from within another.

OpenDoc addresses these issues by enabling the development of a new kind of application, one that provides advantages to both users and developers in the increasingly competitive software markets of today and the future. OpenDoc replaces the architecture of monolithic *conventional applications* in which a single software package is responsible for all its documents' contents, with one of *components* , in which each software module edits its own content, no matter what document that content might be in. This is illustrated in the following figure.



OpenDoc allows developers, large or small, to take a modular approach to development and maintenance. Its component-software architecture makes the design, development, testing, and marketing of integrated software packages far easier and more reliable. Developers can make incremental improvements to products without a complete revision cycle and can get those improvements to users far more rapidly than is possible today.

For developers, this is a radical shift in approach, although its implementation is not difficult. For users, this is only a minor shift in working style; its main effect is to remove the barriers to constructing and using complex documents imposed by conventional monolithic application architecture.

OpenDoc components allow users to assemble customized *compound documents* out of diverse types of data. They also support cross-platform sharing of information. They resolve user frustrations with conventional applications by removing the barriers listed earlier:

- OpenDoc makes it easy for users to assemble any kind of content in any kind of document. OpenDoc documents will accept all kinds of media for which application components exist, now and in the future.
- OpenDoc makes it easy for users to edit any kind of data, in-place, in any document. Users can readily transfer that data to any other document and edit it there just as easily. This lets users focus on document content and lets them take advantage of the context provided by the surrounding document.
- OpenDoc allows businesses to customize their software solutions by assembling components into shrink-wrapped packages, thereby obtaining needed features, increasing reliability, and saving on training costs. In-house developers can then enhance the packages by developing components that integrate smoothly with the off-the-shelf software. Some developers bundle components together into packages that are similar to conventional monolithic applications; others sell individual components for specialized purposes, to users or to other developers for bundling. Users can purchase packages and use them as is, or they can modify a package by adding or replacing individual components.
- OpenDoc lowers market barriers to wide, cross-platform distribution for small developers. Components can be ported easily to other OpenDoc-supported platforms. Small development teams can create individual components with the knowledge that they will integrate smoothly with existing packages created by larger developers.
- OpenDoc promotes collaboration, allowing users simultaneously or separately to build documents as a team, on a single machine or across networks. Documents are not owned by single applications, and changes are recoverable through a draft history that is available for every document. Pervasive scripting support, rich data-transfer capabilities, and an extension mechanism allow users to manipulate their own and other documents in powerful ways.

While providing all of these advantages, OpenDoc exists harmoniously with existing monolithic applications; the user need not abandon conventional applications in order to start using OpenDoc. The following table summarizes some of the principal advantages of the OpenDoc approach to software for both users and developers.

	For Users	For Software Engineering	For Software Marketing	For Small Development Teams
Modularity	Can easily	Can easily test and	Can assemble packages with	Can create components

	add or replace document parts	upgrade components; can reuse components	great flexibility	that work seamlessly with all others
Small size	Less memory and disk space needed	Easier to design, code, debug, and test	Easier, cheaper distribution	Faster development and easier distribution of a component
Multiple platforms	Documents travel across platforms so users can select familiar editors on each	Development effort on one platform can at times be leveraged to others	Opportunities for increased market share	Application of limited resources can at times be used across different platforms
Scriptability and extensibility	Users have greater control over behavior of document parts	Increased ability for communication among parts	Better coordination among components in a package	Increased ability to communicate with other components

---

Overview of OpenDoc

OpenDoc is a set of DLLs designed to facilitate the easy construction of compound, customizable, collaborative, and cross-platform documents. To do this, OpenDoc replaces today's *application-centered* user model with a *document-centered* one. The user focuses on constructing a document or performing an individual task, rather than using any particular application. The software that manipulates a document is hidden, and users feel that they are manipulating the parts of the document without having to launch or switch applications.

This document-centered model does not mean that OpenDoc supports only those kinds of data found in paper documents. An OpenDoc document can contain data as diverse as navigable movies, sounds, animation, database information such as networked calendars, as well as traditional spreadsheets, graphics, and text. OpenDoc is an ideal architecture for multimedia documents. In OpenDoc, each new kind of medium that is developed-video, sound, animation, simulation, and so on- can be represented as part of any document. Thus an OpenDoc document is automatically able to contain future kinds of media, even kinds not yet envisioned, without any modification.

Although OpenDoc lends itself directly to complex and sophisticated layout, its usefulness is by no means restricted to page-layout kinds of applications or even compound documents. The scripting and extension mechanisms allow for communication among parts of a document for any imaginable purpose. Tools such as spelling checkers can be created as components and can then access the contents of any parts in a document that support them; database-access components can feed information to any parts of a document; larger programs such as high-end printing applications can use specialized components to manipulate the data of all parts of a document for purposes such as proof printing and color matching.

The rest of this chapter summarizes the main features of OpenDoc for both users and developers. The rest of this book explains in more detail how to develop software that provides those features.

Parts

OpenDoc uses a few simple ideas to create a structure that integrates a wide range of capabilities. The basic elements are *documents* , their *parts* , their *frames* , and the *part-editor* code that manipulates them. Those elements, represented in a set of object-oriented class libraries, define a number of object classes. The classes provide interoperability protocols that allow independently developed software components to cooperate in producing a single document for the end user. Through the class libraries, these cooperating components share user-interface resources, negotiate over document layout on screen and on printing devices, share storage containers, and create data links to one another.

This section describes part editors, parts and frames, and how parts are categorized, drawn, and stored in documents. It also describes the

two different kinds of part editors and the capability to develop other kinds of OpenDoc components as well.

Documents, Parts, and Embedding

Documents, not applications, are at the center of OpenDoc. Individual documents are not tied to individual applications. In creating OpenDoc documents, collections of software components called part editors replace the conventional monolithic applications in common use today. Each part editor is responsible only for manipulating data of one or more specific kinds in a document.

The user does not directly launch or execute part editors, however. The user works with document parts (or just parts), the pieces of a document that, when executing, include both the document data and the part-editor code that manipulates it. See the following figure.

A page as it normally appears

A page with part boundaries shown

A part can exist as document on its own, or it can be embedded in other parts, allowing the user to create documents of arbitrary complexity.

Parts and the User

In general, a user can perform all the tasks of an application by manipulating a part instead of separately launching or executing a part editor. For example, a user can create and use a spreadsheet part that looks and acts just like a spreadsheet created by a spreadsheet application. However, there are four main differences between a document consisting of parts and a document created by a conventional application:

- An OpenDoc document is built and manipulated differently. OpenDoc users can assemble a document out of parts of any kind-using their graphics part editor of choice, for example, to embed illustrations within a word processor document. Developers can write or assemble such groups of part editors for sale as integrated packages, or users can purchase individual part editors separately.
- New parts of any kind can be added to an OpenDoc document. The user can purchase additional part editors-for example, a charting utility to accompany a spreadsheet-and immediately use them, for example, to embed charts into existing documents.
- In editing, copying, or pasting a part, the user need not be aware of the code that is executing. In fact, the user cannot directly launch a part editor at all. The user manipulates the part data itself, within the context of the document. (It is not necessary to open the part into a separate window for editing). The document containing that part is opened and closed independently of the part's part editor.
- The user can replace part editors. If the editor for a specific kind of part in a document is not available, or if the user prefers to use a different part editor-for example, to replace one charting utility with another-the user can specify that the new editor be used with all parts created under the previous editor. This gives the user the freedom to work with all of the editors of a package or replace any of them with others that the user prefers.

Part Editors

A part editor has fewer responsibilities than a conventional application. Each part editor must:

- Display its part, both on-screen and when printing.
- Edit its part by changing the state of the part in response to events caused by user actions.
- Store its part, both persistently and at run time.

At run time, a part is an object that encapsulates both the state and behavior. The part data provides the state information, and the part editor provides the behavior; when bound together, they form an editable object. As with any object, only the state is stored when the object is stored. Also, multiple instantiations of an object do not mean multiple copies of the editor code; one part editor in memory serves as the code portion for any number of separate parts that it edits.

OpenDoc dynamically links part editors to their parts at run time, choosing an editor based on the kinds of parts that the document contains. Dynamic linking is necessary for a smooth user experience, because any sort of part might appear in any document at any time.

Other Software Components

A user can apply several kinds of OpenDoc components to compound documents. A part editor, as noted earlier, is a full-featured application component; it allows the creation, editing, and viewing of parts of a particular kind. *Part editor*s are the functional replacements for conventional applications; they represent the developer's primary investment. Like applications, part editors are sold or licensed and are legally protected from unauthorized copying and distribution.

*Part Viewer*s are application components that can display a part of particular kind but cannot be used to create or even edit such a part. To enhance the portability of OpenDoc compound documents across machines and across platforms, it is important that part viewers of all kinds be widely available. Developers are expected to create and freely distribute part viewers without restriction for all kinds of parts that they support. A part viewer is really just a part editor with its editing and part-creation capability removed; the developer can create both from the same code base.

Wide availability of a part viewer encourages purchase and use of its equivalent part editor, because users will know that other users will be able to view parts created with that editor. (In some cases it is possible to view a part even when neither its editor nor viewer is present. See the discussion of translation in Part Data Types).

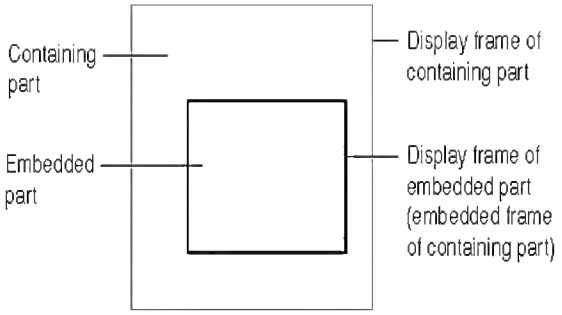
The OpenDoc architecture also allows for the development of software components that, unlike part editors and part viewers, are not directly involved in creating or displaying document parts. Special components called s*ervice*s , for example, provide software services to parts. Spell checking or database-access tools, when developed as service components, can increase the capabilities of part editors. Users could apply them to a single part, to all the parts of a document, or even across separate documents.

Version 1.0 of OpenDoc provides complete support for part editors and part viewers. Future releases may explicitly support services and other types of component software as well.

Frames and Embedding

Parts in an OpenDoc document have a hierarchical arrangement with each other. The logical structure of a document, which underlies its graphical presentation, consists of the e*mbedding* relationships among its parts and their frames.

Parts are displayed in *frame*s , bounded areas that display a part's contents. Frames are commonly rectangular, but may be any shape. The *display frame* of one part-the frame within which the part is viewed-can itself be embedded within the content of another part. The following figure shows such a relationship: the inner frame is an e*mbedded frame* of the outer part, the inner part is an e*mbedded part* of the outer part, and the outer part is the c*ontaining part* of the inner part.



An *embedded part* is logically contained in its containing part, just as its frame is visually embedded in the containing part's frame. The containing part, however, largely ignores the characteristics of its embedded parts. A containing part treats its embedded frames as regular elements of its own content; it can move, select, delete, or otherwise manipulate them, without regard to what they display. The embedded part itself takes care of all drawing and event handling within an embedded frame.

Every document has a single part at its top level, the *root part* , in which all other parts in the document are directly or indirectly embedded. The root part controls the basic layout structure of the document (such as text or graphics) and the document's overall printing behavior. The following figure shows an example of the relationships of the root part and two embedded parts.

- Part 1, a text part, is the root part.
- Part 2, a graphics part, is embedded in part 1.
- Part 3, another text part, is embedded in part 2.

Each embedded part is displayed within its frame, which is embedded at some location in the containing part's content. (See the figure in Activation and Selection for an explanation of the visual characteristics of the frame borders). The following figure shows an example of the relationships of the root part and two embedded frames.

Because it can now contain embedded frames displaying any kind of content, the document is not a monolithic block of data under the control of a single application, but is instead composed of many smaller blocks of content controlled by many smaller software components. In large part, OpenDoc exists to provide the protocols that keep the components from getting in each other's way at run time and to keep documents editable and uncorrupted.

All parts can be embedded in other parts, and every part must be able to function as the root part of a document. However, not all parts are required to be able to contain other parts; simple or specialized utility programs such as clocks or sound players might not have a reason to embed other parts. Such parts are called *noncontainer parts* (or *monolithic parts*); they can be embedded in any part, but they cannot embed other parts within themselves. Parts that can embed as well as be embedded are called c*ontainer parts* . It is somewhat simpler to write an editor for a noncontainer part than for a container part, although container parts provide a far more general and flexible user experience. Unless embedding makes absolutely no sense for your purposes, you should create part editors that support embedding.

Parts can have more than one frame
A part embedded in a document is not restricted to appearing in a single frame. Parts can have multiple frames, displaying either duplicate views or different aspects of the same part. See Presentation for more information.

Part Data Types

Fundamental to the idea of a compound document is that it can hold different types of data. Each part editor can manipulate only its own kinds of data, called its *intrinsic content* . For a simple drawing part, for example, the elements of its intrinsic content might be bit maps. For a simple text part, they might be characters, words, and paragraphs. No part editor is asked to manipulate the content elements of any part whose data it does not understand.

OpenDoc supports both specific and general classifications of data type, so that it can associate parts with specific part editors as well as with general classes of part editors when appropriate.

Part Kind

Different parts in a document hold data of different purpose and different format, understandable to different part editors. For meaningful drawing and editing to occur, OpenDoc must be able to associate a part editor only with data that the editor can properly manipulate.

OpenDoc uses the concept of *part kind* , a typing scheme analogous to file type, to determine which part editor is to be associated with a given part in a document. Because OpenDoc documents are not associated with any single application, a file type is insufficient in this case; each part within a document needs its own "type", or in this case, part kind.

Part kinds are specified as *ISO strings* , (null-terminated 7-bit ASCII strings), although they are usually manipulated as *tokens* (short, regularized representations of those strings) by OpenDoc and by part editors. A part kind specifies the exact data format manipulated by an editor; it may have a name similar to the editor name, such as "SurfDraw" or "SurfBase III", although names more descriptive of the creator and the specific data format itself, such as "SurfCorp:Movie:AVI", are preferable. (The user of such a part would see a different but related string describing the part kind, which in this case might be "SurfTime movie").

Part-editor developers define the part kinds of their own editors. A single editor can manipulate more than one part kind, if it is designed to do so, and a single part can be stored with multiple representations of its data, each of a different part kind. All parts created by an editor initially have its part kind (or kinds), although that can change; see Changing Part Editors.

Part Category

The concept of part kind does not address the similarities among many data formats. For example, data stored as plain ASCII or Unicode text could conceivably be manipulated by any of a number of part editors. Likewise, video data stored according to a standard might be manipulated by many different video editors. Furthermore, text that closely resembles ASCII or Unicode, and stored video that adheres in all but a few details to a particular standard, might nevertheless be editable or displayable to some extent by many text editors or video editors.

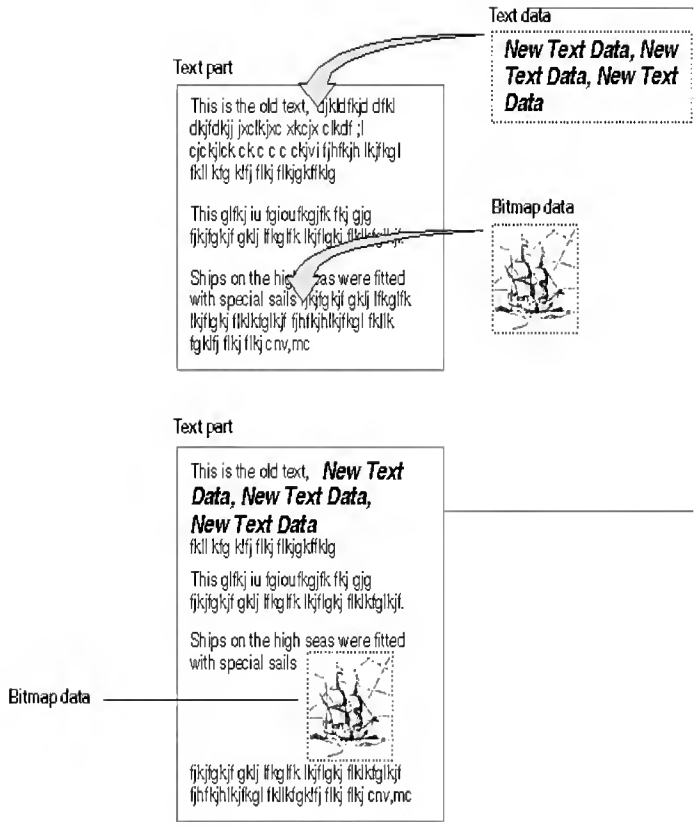
It might be unduly confining to restrict the editing or display of a part of a given kind to the exact part editor that actually created it. Unless the user has every part editor that created a document, the user cannot edit or even view all parts of it. Instead, OpenDoc facilitates the substitution of part editors by defining *part category* , a general description of the kind of data manipulated by a part editor. When users speak of a "text part" or a "graphics part", they are using an informal designation of part category.

Like part kinds, part categories are specified as ISO strings. Part categories have broad designations, such as "plain text," "styled text," "bitmap", or "database." Part-editor developers work with Component Integration Laboratories (CI Labs), a consortium created to coordinate cross-platform OpenDoc development, to define the list of part categories recognized by OpenDoc. See Cross-Platform Consistency and CI Labs for more information on CI Labs. See the table under Part Categories Supported by an Editor for a list of defined part categories.

Each part editor specifies the part categories that it can manipulate. For each defined part category (such as "plain text") the user can then specify a *default editor* (such as "SurfWriter 3.0") to use as a default with any part in that category whose *preferred editor* -the part editor that created it or last edited it-is missing.

Embedding Versus Incorporating

When the user pastes data into a document, the pasted data can either be incorporated into the intrinsic content of the destination part (the part receiving the data), or it can become a separate embedded part. As the following figure shows, the destination part, the part receiving the data, decides whether to embed or incorporate the data. It bases the decision on how closely the part kind and category of the pasted data match the part kind and category of the destination content.



This is how the decision is made:

- If the part kinds of pasted data and destination match, the data should be incorporated; the SurfWriter text shown in the preceding figure, for example, is pasted into the intrinsic data of the SurfWriter (text) part.
- If the part categories are different, the pasted data should be embedded; the SurfPaint bitmap shown in the preceding figure, for example, is embedded in the SurfWriter text part.
- If the part categories are the same but the part kinds are different (a possibility not shown in the preceding figure), the destination part should, if possible, convert the data to its own kind and then incorporate it. Of course, if the destination part cannot read the part kind of the pasted data, it should embed it as a separate part.

In all of these cases, it is the destination part that decides whether to embed or to incorporate. The user can guide that decision by manually forcing a paste to be an incorporation or an embedding; see Handling the Paste As Dialog Box for more information.

Note: For the data being pasted, part kind and part category refer to the kind and category of the outermost, or enclosing, data. There may be any number of parts embedded within that data, of various kinds and categories. Those parts are always transferred as embedded parts, regardless of whether the outermost data is itself incorporated or embedded.

Changing Part Editors

When a part editor stores or retrieves the data of its part, it can only manipulate it using the part kind or kinds that the editor understands and prefers. Furthermore, different systems may have different sets of available part editors. Therefore, the editor assigned to a part can change over the part's lifetime.

For example, the user might open a document (or paste data) containing a part of a kind for which the user does not have the original part editor. In that case, OpenDoc substitutes the user's default editor for that part kind or part category (if the default editor can read any of the part's existing part kinds). If the default editor cannot read the part, OpenDoc searches for an editor that can. Once OpenDoc locates and assigns an editor to the part and the user saves changes, the new editor then becomes the part's preferred editor, and it subsequently stores the part's data using its own part kinds. See Binding for more detailed information on this process.

Lack of a part editor never prevents a user from opening a document. If no editor on the user's system can read any of the part kinds in a part, the part contents remain unviewable and uneditable, and its preferred part editor and part kinds do not change. Nevertheless, OpenDoc still displays a gray box representing the part within the area of the part's frame, and the user may be given the option of translating the data into an editable format, as described next.

Translation

Changing the part editor for a part often means changing data formats and therefore may involve loss of information or may not be directly possible. For example, any text editor might be able to display and edit plain ASCII text without loss, but a sophisticated word processor may be able to read another word processor's data only imperfectly, if at all.

Nevertheless, the user can in some cases choose to employ a part editor with a part that the editor cannot directly manipulate. In such a case, OpenDoc or a part editor performs the necessary translation to convert the part into a format usable by the editor. Translation is possible only if the appropriate translator, or filter, is available on the user's machine to perform the translation between part kinds. The fidelity, or quality, of the translation depends on the sophistication of the translators. Translators are platform-specific utilities that are independent of OpenDoc; OpenDoc simply provides an object wrapper for a given platform's translation facilities.

Note: This function is not supported in the current release of OpenDoc.

OpenDoc provides support for translation when a document is first opened during data transfers such as drag and drop, through semantic events, and at any time the user chooses.

Displaying Parts

In preparing a compound document for viewing, two issues are of prime importance: managing the competition for space among the parts of the document and making provisions for each part to draw itself.

OpenDoc provides a platform-independent interface for part editors, although the interface does not include any drawing commands or detailed graphics structures. OpenDoc provides an environment for managing the geometric relationships among frames, and it defines object wrappers for accessing platform-specific graphic and imaging structures.

Drawing Structures

Each part in an OpenDoc document is responsible for drawing its own content only (including, at certain times, the borders of the frames embedded within it). Thus, a part does not draw the interiors of its embedded frames because they contain the content of other parts (which must draw themselves).

Drawing a document is therefore a cooperative effort, for which no part editor is completely responsible; each part editor is notified by OpenDoc when it must draw its own part.

Drawing in OpenDoc relies on three fundamental graphics-system-specific structures for which OpenDoc provides these related objects:

- **Canvas**, a description of a drawing environment.

  A *dynamic canvas* , such as a screen display, can potentially be changed through scrolling or paging.

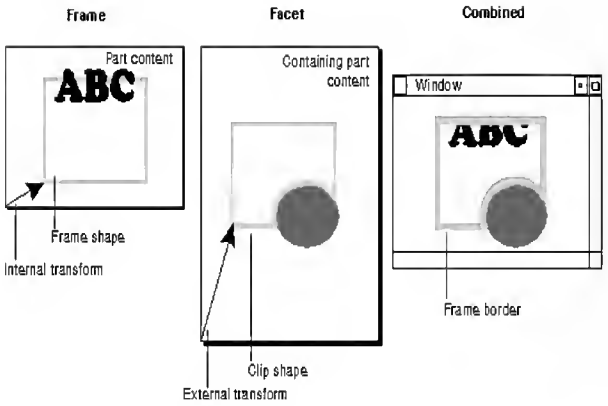
  A *static canvas* , such as a printer page, cannot be changed once it has been rendered. OpenDoc allows for different behavior when drawing onto dynamic and static canvases.

- **Shape**, a structure that describes a geometric shape.

- **Transform**, a structure that describes a standard set of 2-dimensional transformations, such as an offset, scaling, or rotation.

When a part actually draws itself within a frame, it uses an object closely related to the frame. A facet is an object that is the visible representation of a frame (or a portion of a frame) on a canvas. A frame typically has a single facet, but for offscreen buffering, split views of a part, or special graphics effects, it may have more than one.

In general, frames control the geometric relationships of the content that they display, whereas facets control the geometric relationships of frames to their containing frames or windows. The facet is associated with a specific drawing canvas, and both frames and facets have associated shapes and transforms. The following figure summarizes some of the basic relationships among frames, facets, shapes, transforms, and canvas in drawing; for more detail see Transforms and Shapes.



The left side of the preceding figure shows the content area of an embedded part as a page with text on it. The portion of the part that is available for drawing is the portion within the *frame shape* assigned to the part by its containing part. (A part may have more than one frame, but only one is shown here). The frame's *internal transform* positions the frame over the part content. (Another shape, the *used shape* , defines the portion of the frame shape that is actually drawn to; in this case, the used shape equals the frame shape).

The center of the preceding figure shows the facet associated with this embedded part's frame. (A frame may have more than one facet, but only one is shown here). The c*lip shape* defines where drawing can occur in relation to the content of the containing part. In this case, the containing part is the root part in a document window, but it could be an embedded part displayed in its own frame. The clip shape is typically similar to the frame shape except that, as shown in the preceding figure, it might have additional clipping to account for other parts or for elements of the containing part that overlap it. The facet's e*xternal transform* positions the facets within the containing part's frame and facet. (Another shape, the *active shape* , defines the portion of the area of the facet that the embedded part will respond to events within. in this case, the active shape equals the frame shape).

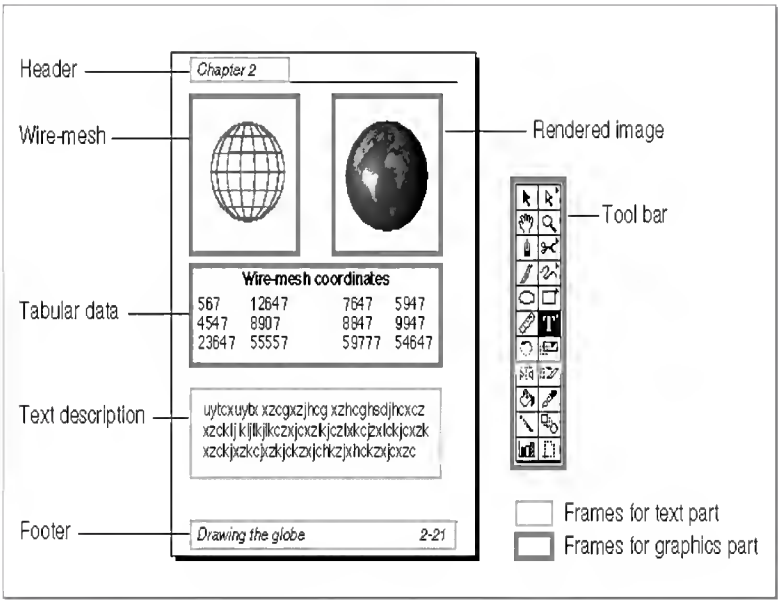
The right side of the preceding figure shows the result of drawing both parts on the window's canvas. The portion of the embedded part defined by the frame is drawn in the area defined by the facet. If this embedded part is the *active part* , meaning that the user can edit its contents, OpenDoc draws the *active frame border* around it; the shape of that border is based on the facet's active shape.

Presentation

There are many ways that a part editor can draw the contents of a part. A word processor can draw its data as plain text, styled text, or complete page layouts; a spreadsheet can draw its data as text, tables, graphs, or charts; a 3-D drawing program can draw its shapes as wire-mesh polyhedrons, filled polyhedrons, or surface-rendered shapes with specified lighting parameters.

For any kind of program, individual frames of a single part can display different portions of different views of its data. For example, in a page layout part, one frame of a page could display the header or footer, while another could display the text of the page, and yet another could show the outline of the document. For a 3-D graphics part, different frames could show different views (top, front, side) of the same object. For any kind of program, an auxiliary palette, tool bar, or other set of controls might be placed in a separate frame and be considered an alternative "view" of the part to which it applies.

OpenDoc calls such different part-display aspects part *presentations* and imposes no restrictions on them. The following figure shows some examples of different presentations for individual parts in a document. There are only three embedded parts, but two of them have several display frames, each with a different representation.

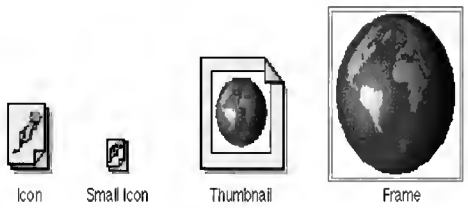


Your part editor determines the presentations that its parts can have, and it stores a presentation designation (for your own drawing functions to make use of) in each of your part's frames. You can store other display-related information as the *part info* data of a frame. Each frame and each facet has the capability to store part info data that you can access; in that data, you can store any useful private information relating to the display of your part in that frame or facet. Only you use the part info data of your frames and facets.

View Type

OpenDoc does not specify presentation type, but it nevertheless defines some aspects of part display. Each part has a *view type* , a designation of its fundamental display form of the part in that frame. Basically, view type states whether a part is to be displayed in icon form or with its contents visible.

In most situations in most documents, each part displays its contents, or some portion of its contents, within its frame. However, a part can also display itself as one of several kinds of icons. See the following figure for examples.



Each basic display form (standard icon, small icon, thumbnail icon, or framed) is a separate view type. Your part editor stores a designation of the part's view type in each of your part's display frames. A single part can, of course, have different view types in different frames.

On the desktop and in Workplace Shell folders, parts are individual closed documents and by default have an icon view type. When such a document is opened, the part gives its frame a frame view type, and its contents (including embedded parts) then become visible. Nevertheless, open documents can have embedded parts displayed as icons. Keep in mind that a part with an icon view type is not necessarily closed or inactive. A sound part, for example, might have nothing but an icon view type, even when playing its sound.

Each containing part specifies, by setting a value in its embedded part's frame, the view type it prefers the embedded part to have. Your container parts can specify the view type for each embedded frame they create; when embedded, your parts should initially display themselves in the view type expected by their containing parts. See, for example, View Type for more information and guidelines.

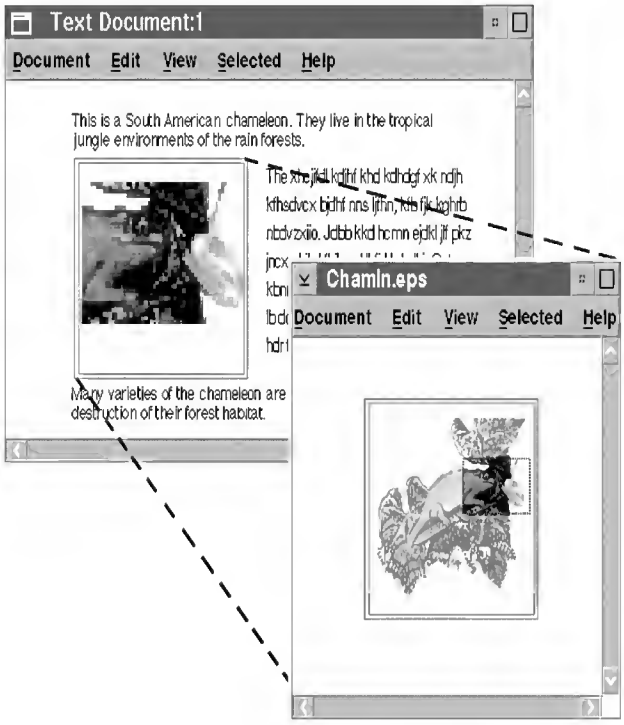
Document Windows and Part Windows

Compound documents are displayed in windows. A *document window* is a window that holds a single OpenDoc document; that document

can contain a single part or many parts. Document windows in OpenDoc are essentially the same as the windows that display a conventional application's documents.

An architectural cornerstone of OpenDoc is that it provides *in-place editing* of all parts in a compound document. Users can manipulate the content of any part, no matter how deeply embedded, directly within the frame that displays the part.

Nevertheless, a user might wish to view more of a part than is displayed within a frame. Even if a frame is resizable and supports scrolling of its contents, it might be more convenient to view that frame's part separately, in its own window. OpenDoc supports this by allowing users to open a separate window, called a *part window* , which looks similar to a document window. See the following figure for an example.

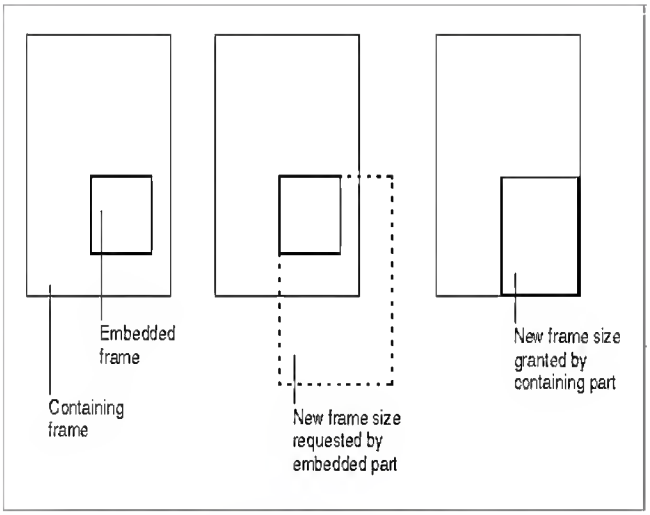


Like document windows, part windows can themselves contain embedded parts. But because your part is the root part in any part windows that your part editor creates, you may take a more active role in window handling than you do for your part when it is an embedded part in a document window. See Creating and Using Windows for information on handling part windows.

Frame Negotiation

In a compound document with embedded parts, the embedding hierarchy and the frame locations determine the geometric relationships among parts. Each part controls the positions, sizes, and shapes of the frames embedded within it. If an embedded part needs to change the size of its frame or add another frame, it must negotiate for that change with its containing part. This *frame negotiation* allows an embedded part to communicate its needs to its containing part; however, the containing part has ultimate control over the embedded frames' sizes and shapes.

The following figure shows a simple example of frame negotiation. A user edits an embedded part, adding enough data so that the content no longer fits in the embedded part's current frame size. The embedded part requests a larger frame from the containing part. The containing part can either grant the request or return a different frame size from that requested. In this example, the containing part cannot accommodate the full size of the requested frame and so returns a frame size to the embedded part that is larger than the previous frame but not as large as the requested one.



The frame-negotiation process is described in more detail in Frame Negotiation.

Event Handling

Most part editors interact with the user primarily by responding to user events. *User events* are messages sent or posted by the operating system in response to user actions, activation or deactivation of a part or window, or messages from other event sources. Based on information in a user event, a part editor might redraw its part, open or close windows, perform editing operations, transfer data, or perform any sort of menu command or other operation.

OpenDoc has several built-in event-handling features that help your part editor function properly within a compound document. For example, instead of polling for events, as a typical application does, your part editor acts when notified by OpenDoc that an event has occurred.

This section notes some of OpenDoc's event-handling features; for more information on user events, see User Events.

The Document Shell and the Dispatcher

Part editors respond differently to user events than conventional applications do. Part editors do not receive events directly; OpenDoc receives them and dispatches them to the proper part.

Because they are not complete applications, and because they must function cooperatively, part editors run in an environment that itself handles some of the tasks that conventional applications typically perform. That environment is called the OpenDoc *document shell* ; it is an executable that handles certain application-level tasks and provides an address space for each OpenDoc document within which part editors manipulate document content.

Whenever an OpenDoc document is opened, OpenDoc creates an instance of the document shell. The shell creates global objects and uses them to open the document. OpenDoc then loads the part editors for all parts that appear in the document window; the part editors read in the data of their own parts. The shell receives both user events and scripting-related events (see Scripting Support). The shell uses the OpenDoc *dispatcher* to dispatch those events to the proper part editors, based on event location and ownership of shared resources such as menus.

The document shell is described in The Document Shell. How the dispatcher sends events to parts is described in How User Events Are Handled.

Handling User Commands

As a result of user actions or commands, OpenDoc interacts with part editors to perform these common application activities:

- Part activation
- Menu handling
- Undo

In general, the document shell receives events and passes the appropriate information to the proper part.

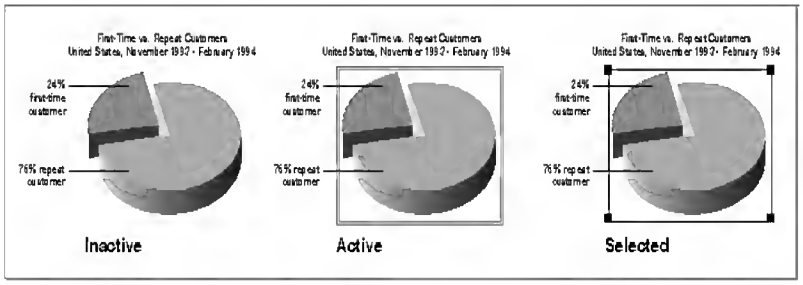
Activation and Selection

In response to user actions, individual parts in a document become active and thus editable. A part is *active* if it possesses the s*election focu*s ; the user can select and modify its contents in this state. The active part may also possess the menu and keystroke focus; see Focus Transfer for more information on foci.

Parts activate themselves in OpenDoc, and the individual parts in a document must cooperate to transfer the selection focus among each other as appropriate. Note that when a part is not in the active (editable) state, it need not be idle; multiple parts within an OpenDoc document can perform different tasks at the same time.

Switching among individual parts within a document can involve a much less intrusive context switch than switching from one conventional application to another. Users are less likely to be irritated, because the wait before they can edit a newly activated part is not all that perceptible.

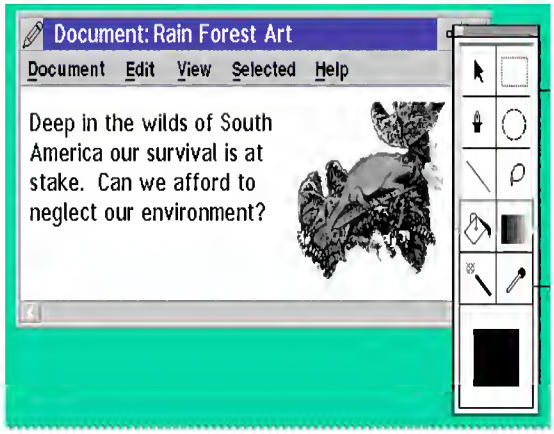
The active state is different from the selected state. When a part is s*elected* , its *frame* is made available for manipulation. Because embedded frames are considered to be content elements of their containing part, they can be selected and then moved, adjusted, cut, or copied just like text, graphic objects, or any other content elements. Thus, whereas an active part is manipulated by its own part editor, a selected part is manipulated-as a frame-by its containing part. The following figure shows the visual differences among the inactive, selected, and active states of an embedded part.



Note that an *inactive* part, one that is not being edited, need not have a visible frame border. A selected part's frame border is drawn by the containing part; its shape typically corresponds to the frame shape, and its appearance should follow guidelines for selected frames. An active part's frame border is drawn by OpenDoc; its shape corresponds to the active shape of the embedded part's facet, and its appearance is fixed for each platform.

When a part activates, more than its own frame border changes. The following figure illustrates some of the visual changes to a window caused by part activation.

1. Text part active



2. Graphics (root) part active

In this figure, the text part is active first. OpenDoc has drawn the active frame border around its frame, it has a highlighted selection, and it displays its own menus. Then the text part becomes inactive and the graphics part becomes active. The text part removes its menus and its highlighting. The graphics part displays its own menus and also displays two palettes, in separate windows. (OpenDoc in this case does not draw the active border around the graphics part's frame, because the graphics part is the root part.)

Activation generally occurs in response to mouse clicks within the area of a frame (specifically, within the area of a facet's active shape). OpenDoc follows an *inside-out activation* model, in which a single mouse click causes activation of the smallest (actually, the most deeply embedded) enclosing frame at the pointer location. In the case of a deeply nested embedded frame, as shown in the following figure, a single click within the frame of the most deeply embedded part activates that part and allows the user to start editing immediately.

By contrast, some compound-document architectures use an *outside-in activation* mode, in which the user selects the outermost (nonactive) frame with one click and activates it with the second click. Thus, many clicks might be necessary for the user to activate a deeply embedded part. In the preceding figure, for example, the user would have to click four times to start editing.

Despite the advantages of inside-out activation, a containing part may at times not want an embedded part to be activated when the user clicks within its frame; it may instead want the part to become selected. For example, when you want the user to be able to move-but not edit-a part embedded within your part, you would rather have it selected than activated by a mouse click. OpenDoc allows you to specify this behavior for an embedded part by placing its frame in a *bundled* state. A bundled frame acts like a frame with outside-in selection, in that a single click selects the frame. (However, unlike with outside-in selection, subsequent clicks do not then activate the part; you have to unbundle it before it can be activated).

Your part does not receive window-activation events directly. The OpenDoc dispatcher notifies your part when its window becomes active or inactive, as described in Handling Activate Events. It also notifies the containing part, rather than the embedded part, when an embedded part

should become selected.

Menus

Menu organization and menu handling are different on different platforms. OpenDoc encapsulates certain aspects of menu behavior in a platform-neutral menu bar object. That object gives your part editor access to its own and to OpenDoc's menus.

There are four menus that are available whenever an OpenDoc document is active. It is recommended that these menus remain unchanged by the part editor, with the exception of adding more choices. The part is free to put up as many other menus as desired by the developer.

The four standard menus are:

- The Document menu is basic to the OpenDoc user interface. It provides a replacement for the application-oriented File or equivalent menus. The contents of the Document menu reflect the OpenDoc document-centered approach. For example, there is no **Quit** or **Exit** command. In OpenDoc, the user closes a document rather than quits applications.

  The Document menu contains choices that affect the top-level part shown in the window.

- The Edit menu is another standard OpenDoc menu that allows the user to operate on your part or its contents when it is the active part.

  This menu contains standard choices on selected parts or selected intrinsic data. For example, there are options to create a new part of the same type as the selected part and to undo and redo user actions.

- The View menu contains choices that affect the view of the active part. It also controls the display of related windows, such as tool bars or palettes. All container parts should support the Icon, Tree, and Details views.

- The Help menu contains choices that provide help on the active part.

In addition to the menu bar, each part has a pop-up menu just like other objects in the WorkPlace Shell. The only difference between the pop-up menu and choices in the menu bar menu is that in pop-up menus unavailable choices are not shown. In menu bar menus, unavailable choices are grayed.

More information on the individual menus is in Menus.

Most of the items in the Document menu are handled by the document shell; see The Document Shell and the Document Menu and Document Menu for more information.

Your part editor can add menus of its own, and it can modify-within strict limits and according to certain rules-the standard OpenDoc menus.

Undo

Applications on many platforms provide a form of *undo* , a capability that allows a user to reverse the effects of a recently executed command. OpenDoc provides better support for undo than do most current platforms, in at least two ways:

- The undo action can cross document boundaries. This is important because a single drag-and-drop action can affect more than one part or document.

- OpenDoc allows multiple sequential undo actions. The user can undo multiple sequential commands, rather than only one.

OpenDoc support for undo is described in more detail in Implementing Undo.

Storage and Data Transfer

All the data of all parts in a document, plus all information about frames and embedding, is stored in a single document file. OpenDoc does not require the user to manually manage the various file formats that make up a compound document; OpenDoc manages and holds all the pieces. This makes storage easier for developers and exchanging documents easier for users.

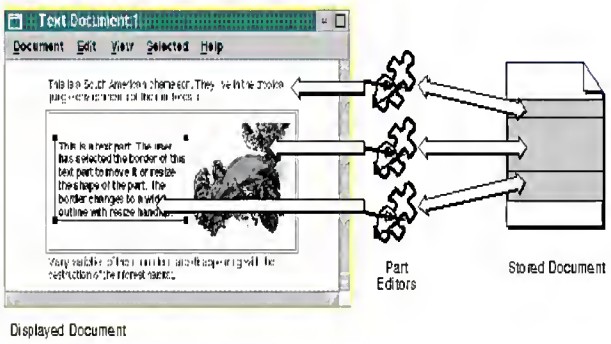
OpenDoc uses the same data-storage concepts for data transfer (clipboard, drag and drop, and linking) that it uses for document storage.

Storage Basics

The OpenDoc storage system manages persistent storage for parts. It is a high-level persistent storage mechanism that enables multiple part editors to share a single document file effectively. The storage system is implemented on top of the native storage facilities for each platform that supports OpenDoc.

Storage in OpenDoc is based on a system of structured elements, each of which can contain one or more data streams. The data of each part in a document is kept in at least one storage unit, distinct from the data of other parts. Storage units can also include references to other storage units, and OpenDoc uses chains of such references to store the embedding relationships of the parts within a document.

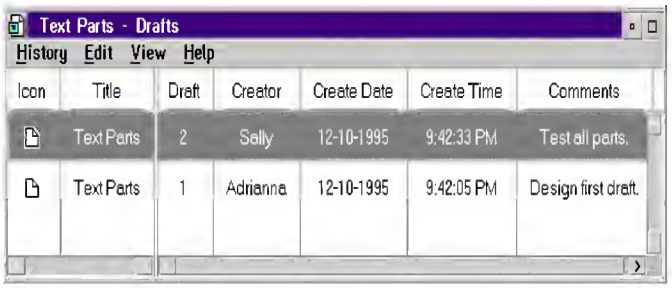
Storage units and other elements of structured storage of OpenDoc are described further in Storage.



Document Drafts

OpenDoc documents have a history that can be preserved and inspected through the mechanism of drafts. A *draft* is a captured record of the state of a document at a given time; the user decides when to save the current state of a document as a new draft and when to delete older drafts. All drafts are stored together in the same document, with no redundantly stored data.

The OpenDoc draft mechanism helps in the creation of shared documents. When several users share a document, each in turn can save the current state of the document as a draft and then make any desired changes. Users can always look back through the drafts of the document they and others have created. Also, if translation occurs during the process of sharing documents, the user can consult an older draft to regain access to formatting information that might have been lost in translation. The following figure shows an example of a dialog box through which the user can manipulate drafts.



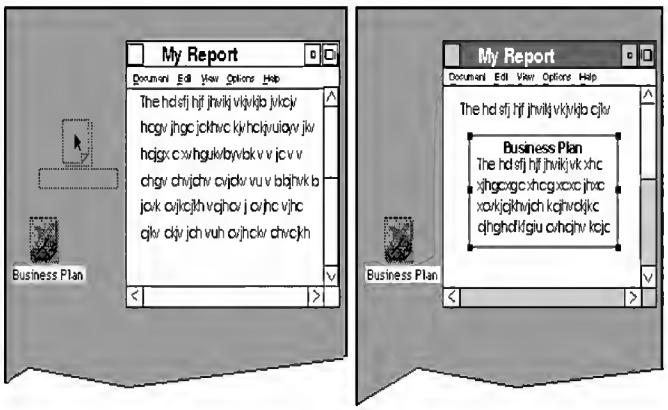
Parts also interact with their drafts to create the objects needed for embedding other parts and to create links to data sources. See Drafts for more information.

Templates

OpenDoc gives the user additional aids for constructing compound documents. One aid, central to the user's ability to create new kinds of parts, is templates.

Templates are specialized parts or documents used in creating other parts. A template is never opened; when the user attempts to open a template, a copy of that part is created and opened instead. Users can create templates with specific formatting and content, to create letterhead, forms, or types of documents. Template parts can be embedded in documents, or they can exist as stand-alone documents themselves.

The following figure shows an icon for a template document on the desktop. The user drags the template icon and drops it onto the document, at which time a copy of the template part is embedded in the document and opened into a frame, displaying the part's initial contents.



1 2

It is typically through templates that users first gain access to your part editor. When you develop and ship a part editor, you can also provide one or more template documents. Part registration creates a default template as well. To create a part using your part editor, the user does not launch the editor; instead, the user double-clicks on your template document or drags it into an open document window.

Data Transfer

OpenDoc includes several built-in data-transfer features that allow users to create and edit OpenDoc documents more easily than is usual with conventional applications. Users can put any kind of media into a document with simple commands, and OpenDoc helps your part editor respond to those commands.

In OpenDoc data transfer, the source is the part (or the portion of its content) that provides the data being transferred, and the destination is the part (or the location in its content) that receives the transferred data.

When source data in one format is transferred to a destination whose data is written in another format, translation may be necessary; see Translation. Translation usually involves loss of data, and is less necessary if part editors provide standard content formats, as promoted by CI Labs; see Cross-Platform Consistency and CI Labs.

Clipboard

Clipboard data transfer allows for easy exchange of information among documents, using menu commands familiar to most users. OpenDoc supports clipboard transfer of any kind of data, including multipart compound data, into any document.

Clipboard transfer is a two-stage process. The user first selects some portion of the content of the source part (possibly including embedded parts) and places a copy of that content onto the clipboard buffer by executing the Cut or Copy command. At any subsequent time, the user can copy the clipboard data to the destination (back into the same part, into another part in the same document, or into another document) by executing the Paste command.

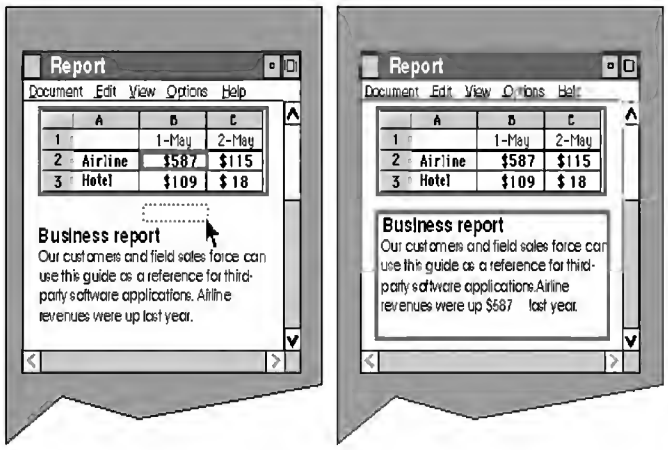
As noted in Embedding Versus Incorporating, OpenDoc allows for an intelligent form of pasting in data transfer, anticipating user expectations about the result of a pasting operation when the destination holds a different kind of data from the source. Subject to user override, the destination part can decide whether to embed the data as a separate part, or incorporate it as content data into itself.

Clipboard transfer is discussed in detail in Handling Pasted or Dropped Data.

Drag and Drop

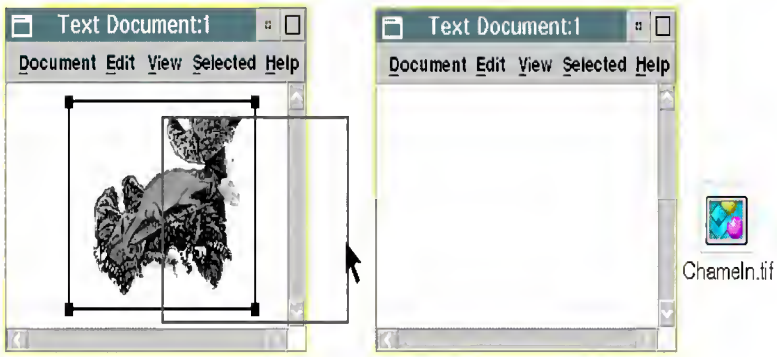
Drag-and-drop data transfer is similar to Clipboard transfer, except that it involves direct user manipulation of the data being transferred, rather than the intermediate use of the Clipboard. OpenDoc supports drag and drop within documents, across documents, and to the desktop. Users can even drop non-OpenDoc data into OpenDoc parts.

The following figure shows the use of drag and drop to transfer a piece of information from a spreadsheet part to a text part within a document. As with Clipboard transfer, OpenDoc uses intelligent pasting in drag and drop; the spreadsheet part includes a plain-text version of the selection being dragged, so that the destination part (the text part) can directly incorporate it at the location of the drop.



1. 2.

Drag and drop works equally well between separate documents. As far as user experience is concerned, the data-transfer facilities of OpenDoc actually blur the distinction between a part and a document. If the user drags a closed document (represented as an icon on the desktop) into an open document window, a copy of the transferred document either becomes an embedded part in the window's document or is incorporated into the intrinsic content of the document's root part. Likewise, if the user selects the frame of an embedded part in an open document window and drags or otherwise moves that part to the desktop, it immediately becomes a separate, closed document represented by icon as shown in the following figure.



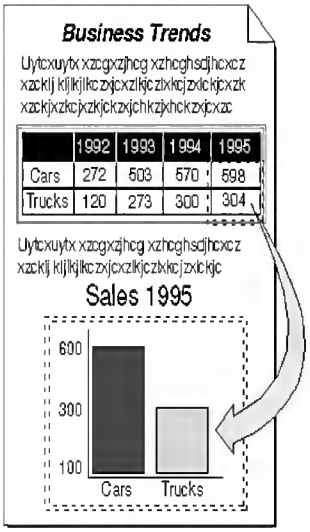
1 2

Drag and drop is discussed in detail in Drag and Drop.

Linking

Linking allows the user to view, within a part's frame, data that is a live (updatable) copy of data in a different location-in the same frame, in a different frame, in a different part, or even in a different document. When the data in the source location changes, OpenDoc updates the copy at the destination, either automatically or manually, depending on user preference.

Linked data can include embedded parts, and the source of a link can be in the same part as its destination, in a different part in the same document, or in an entirely different document. The following figure shows a simple example of linking between two parts in a document. In this example, whenever the user changes the values in the linked spreadsheet cells, the bar graph adjusts its display accordingly.



Users create a link when pasting data, either by selecting Paste Link or Paste As. The Paste As dialog box is used to link the source of the data to its destination. (See The figure in Handling the Paste As Dialog Box for an example of the dialog box). This simple user interface to linking makes the use of links more attractive to users than some other systems do.

Linking is discussed in detail in Linking.

Extensibility

The OpenDoc architecture is designed to be extended. Using built-in features, you can enhance the capabilities of and communications among your parts in a compound document, and you can even develop component software that goes well beyond the standard OpenDoc model of parts and compound documents.

The main point of departure for enhancing OpenDoc is the extension mechanism, a general method for adding programming interfaces to objects. Additional mechanisms are the focus-module and dispatch-module interfaces, which allow you to add new kinds of focus and new kinds of event handling to your part editors.

OpenDoc already includes three instances of extensions: scripting support, the settings extension, and the view extension. This section introduces the features of these extensions and summarizes how you can add other extensions for other purposes.

Scripting Support

A major feature available with OpenDoc is the ability to make parts scriptable, allowing users to customize their applications to user-specific tasks. Working in a compound document environment, with scriptable parts, sophisticated users can use standard document-editing procedures to add application-like functionality to parts. Likewise, programmers can create complex client applications and present them in the form of compound documents.

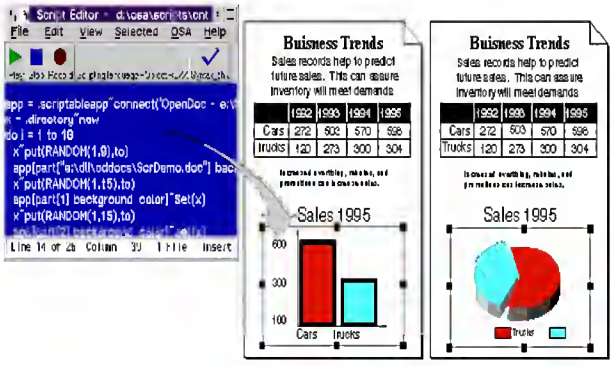
The scripting supported by OpenDoc is called *content centered scripting* or scripting based on a *content model* , which lists the *content objects* and operations that the part makes available for scripting. For a part to be scriptable, its part editor must have a content model. OpenDoc provides a way to deliver scripting messages, or s*emantic event*s , from a scripting system to a part editor, which responds to events. Object REXX is a scripting language based on *Open Scripting Architecture* .

Your part editor can provide increasing levels of scripting support, including s*criptability, recordability* , and *customizability* . They are described in Semantic Events and Scripting.

When a scripting action involves data transfer, OpenDoc can decide whether to embed the transferred data as a separate part or translate it and incorporate it as content data into the destination part. Translation and embedding versus incorporation are described in Translation.

Scripts can be attached to parts, and a user-interface control such as a button might consist of nothing more than a part that executes a script when activated. Script editors also can send semantic events to parts. The following figure shows an example of a script editor sending a semantic event based on a scripting command ("Set chartType of part "sales chart" to type3D pie") to a document. The document shell decodes the *object specifier* in the event (part "sales chart") and determines the content object that it applies to (in this case an embedded part). The embedded part then performs the specified operation.

The scripting support in OpenDoc is an enhancement of OpenDoc's basic capabilities. It is recommended, but not required, that a part editor support scripting.



Settings Extension

The settings extension allows a part to display its properties and settings in a standard OS/2 notebook. OpenDoc provides Property pages that are added to the notebook by the part.

View Extension

The view extension allows a part to communicate with the Details view of its container part and supply information for additional data columns in the view container.

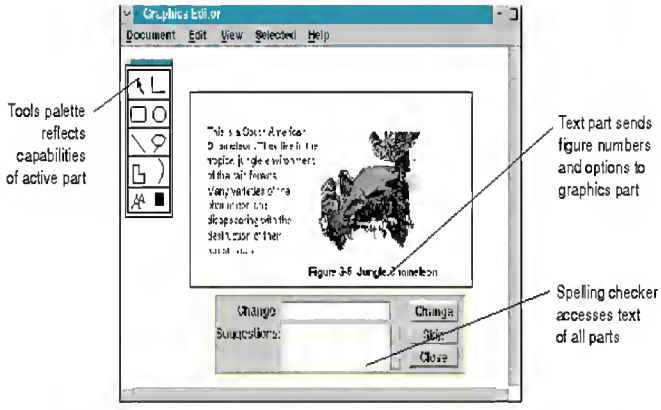
Other Extensions

The basic architecture of OpenDoc is primarily concerned with mediating the geometric interrelationships among parts, their sharing of user events, and their sharing of storage. Direct communication among parts for purposes other than frame negotiation is mostly beyond the basic intent of OpenDoc. If separate parts in a document (or across documents) need to share information, or if one part needs to manipulate the contents or behavior of another part, some extension to the capabilities of OpenDoc is needed.

OpenDoc allows you to add these kinds of capabilities to part editors through its extension mechanism. A part editor can create an associated *extension* object that implements an additional interface to that editor's parts. Other part editors can then access and call that interface.

The OpenDoc support for scripting is an example of an extension interface, as noted in the previous section. Other extension interfaces can, however, provide faster interaction and lower-level communication between parts than is possible by passing semantic events. A word processor, for example, might construct an extension interface to give related part editors or other tools access to its text. A developer of a large application package might use extensions to provide greater integration among its components.

The following figure shows some possible examples of communication through OpenDoc extensions. It shows a text part and a graphics part that both communicate with a tool palette (a third part), so that the palette displays the appropriate tools for the currently active part, and communicates the user's tool selection back to the active part. A spelling checker provides its service to all parts in the document and accesses their text through their extensions. Furthermore, the text part and graphic part communicate directly through an extension mechanism; the text part updates the graphic part with the current figure number and caption it has assigned to the figure.



In addition to the scripting extension, OpenDoc includes existing extensions and other kinds of enhancements that allow you to extend the OpenDoc Properties notebook, extend the capabilities of the OpenDoc document shell, or provide for additional kinds of user events or foci. Use of the OpenDoc extension mechanism is discussed further in Extending OpenDoc.

Cross-Platform Consistency and CI Labs

OpenDoc was designed from the beginning to be a cross-platform architecture. The programming interface to OpenDoc is general enough that it is readily implemented on many platforms, adapting well to different user-interface designs and different run-time models. Although OpenDoc does not provide a complete programming interface-it does not, for example, replace the graphics system or drawing commands for any platform-it provides consistent structure within which such system-specific interfaces can be used. As a result, users on all platforms obtain a uniform experience in embedding and manipulating all kinds of media.

Part of this uniformity is built into OpenDoc, and part comes from additional platform-independent user-interface guidelines that part-editor developers must follow. Because platform-neutral user-interface guidelines and programming standards are so important, and because data integrity and data-transfer protocols are so critical in cross-platform settings, OpenDoc is affiliated with an organization devoted to solving these issues in a vendor-independent fashion.

Component Integration Laboratories, Inc. (CI Labs) is a nonprofit association dedicated to promoting the adoption of OpenDoc as a vendor-neutral industry standard for software integration and component software. CI Labs is composed of a number of platform and application vendors with a common interest in solving OpenDoc issues and promoting interoperability.

Organizations and individuals who want to participate in the move to a component-software model are invited to join CI Labs. CI Labs supports several levels of participation through different membership categories. Specific membership benefits vary by category, but all members influence the future direction of OpenDoc technology.

We encourage you to add your name to one of our electronic mail information lists at CILABS.ORG, download files from our server at FTP.CILABS.ORG, or look up our Web page (http://www.cilabs.org). If for some reason you are unable to get files from our server, we can send you an information packet. For details on membership levels and how you can become a member, please provide the following information:

Name
Company Name
Title
Street Address
City
State/Province
Zip Code/Postal Code/Country
Telephone Number
Fax Number
e-mail address

You can send it through CILABS@CILABS.ORG or use our US mail address:

Component Integration Laboratories, Inc.
PO Box 61747
Sunnyvale, CA 94088-1747

Telephone	(408) 864-0300
FAX	(408) 864-0380
Internet	CILABS@CILABS.ORG
World Wide Web	http://www.cilabs.org

CI Labs is responsible for the development and maintenance of a number of component software packages, including the following:

- **OpenDoc**, the platform-independent class library and compound document architecture that is described in this book.
- **Bento**, a class library and format for persistent object storage. Bento allows OpenDoc to store and exchange compound documents and multimedia.
- **Open Scripting Architecture (OSA)**, a scripting and program-automation standard and interface that supports application-independent user scripting on multiple platforms. OSA is not a scripting language; it is a messaging specification and library upon which scripting languages are built.
- **System Object Model (SOM)**, an efficient and powerful mechanism and set of libraries for dynamic linking of objects. SOM underlies OpenDoc's dynamic linking capability, and its language-neutral interface compiler permits development in multiple languages, object-oriented as well as procedural, even within a single software component.
- **Component Glue,** an interface and library that enables seamless interoperability between OpenDoc and Microsoft Corporation's Object Linking and Embedding (OLE) technology for interapplication communication.

CI Labs oversees the development and distribution of these technologies. In the future, CI Labs also expects to coordinate the following activities for developers of OpenDoc part editors:

- Certification of part editors for consistency with OpenDoc standards and for safe use in any OpenDoc document
- Promotion of standard formats for exporting different kinds of document content
- Coordination of standards for extending the capabilities of OpenDoc, by defining specific sets of OpenDoc extension interfaces

---

Development Overview

Creating OpenDoc software is not difficult, but it represents a shift in approach from conventional application development. OpenDoc part editors differ from conventional applications in that:

- They are generally smaller than conventional applications.
- They do not have direct access to some operating-system services such as event dispatching.
- They do not own their own documents.
- They must work in close cooperation with other part editors.

The cross-platform design of OpenDoc means that some aspects of part development may seem foreign to developers, but it also means that writing part editors for multiple platforms is far easier than doing so for conventional applications.

OpenDoc is an object-oriented system, but a part editor can be written in procedural code and still fit into the OpenDoc class structure. Existing applications, object-oriented or not, can be retrofitted easily to work in the OpenDoc environment. OpenDoc is extensible, and many of its components are replaceable, allowing for innovation by developers at both the system and application levels.

This chapter introduces the OpenDoc class library, and presents a high-level summary of several approaches to developing an OpenDoc part editor. For additional discussions of general development-related issues, see also OpenDoc Run-Time Features.

The OpenDoc Class Library

OpenDoc is a set of DLLs with a largely platform-independent, object-oriented programming interface. This section introduces the classes implemented in the OpenDoc libraries and the design goals behind them.

The interfaces to all of OpenDoc's classes are specified in the *Interface Definition Language (IDL)* , a programming-language-neutral syntax for creating interfaces. IDL is part of the System Object Model (SOM), a specification for object binding at runtime. IDL interfaces are typically compiled separately from implementation code, using a SOM compiler. See Developing with SOM and IDL for more information.

Because OpenDoc uses IDL and SOM, part editors and other OpenDoc objects that have been created with different compilers or in completely different programming languages can nevertheless communicate properly with each other. Furthermore, they can be independently revised and extended and still work together.

For more complete information on the OpenDoc class library, including detailed descriptions of all public OpenDoc classes and methods, see the *OpenDoc Programming Reference* .

A Set of Classes, Not a Framework

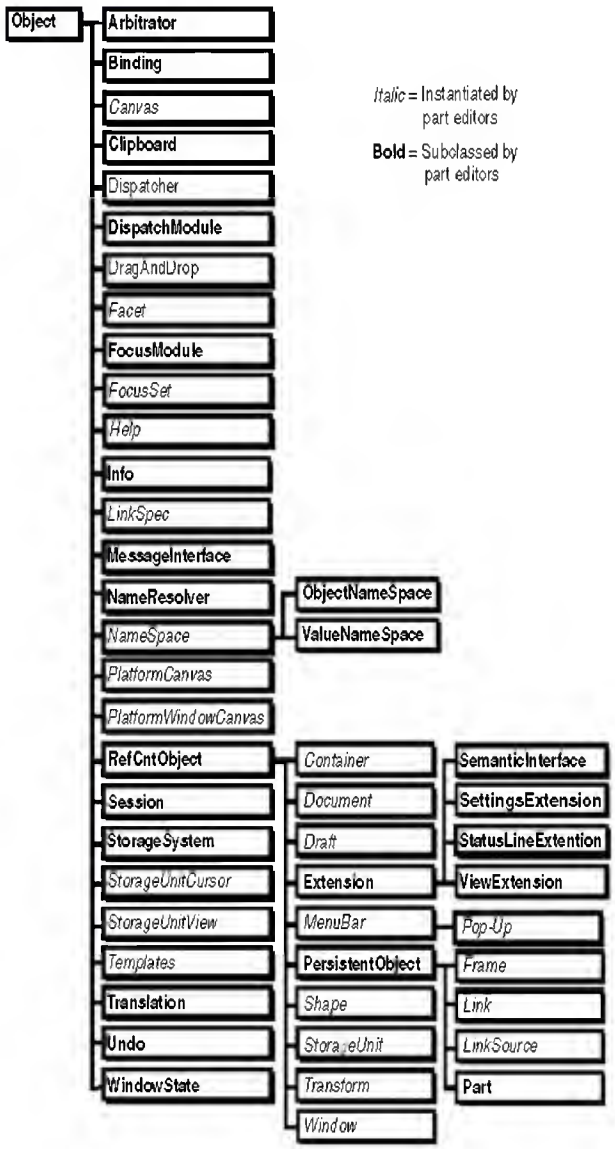
OpenDoc is a set of classes whose objects cooperate in the creation and manipulation of compound documents. It is designed to be as platform-neutral as possible. The object-oriented library structure, in which object fields are private, facilitates the replacement of existing code in a modular manner. Also, by using abstract classes and pure virtual methods, it defines a structure for part editors while specifying as little as possible about their internal functioning.

The OpenDoc class library is not an object-oriented application framework in the full sense of the word. That is, even though the design of OpenDoc allows you to create a part editor, it does so at a lower level, and without forcing as much adherence to interface and implementation as is typical for an application framework.

You can create a part editor just as effectively in either way: directly to the OpenDoc library interfaces or indirectly through the interfaces of a framework. The main difference is that if you use a part-editor framework, you have less code to actually implement yourself, and it is easier to ensure consistency in the final product.

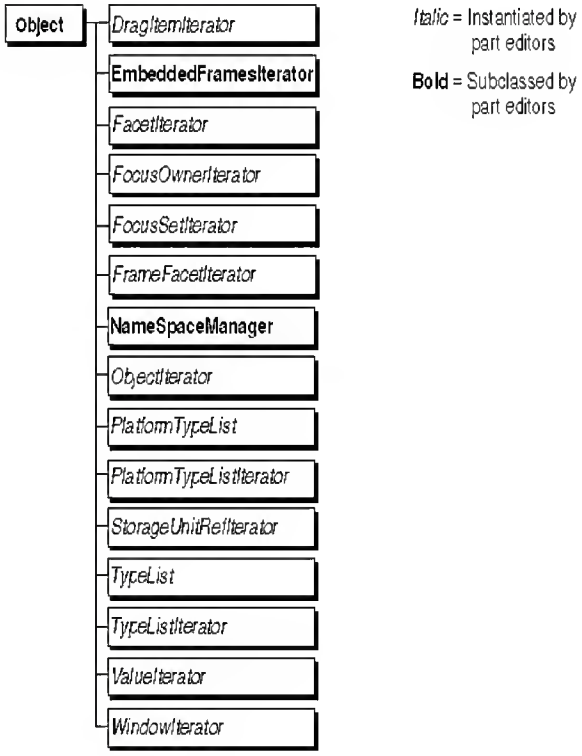
The principal classes in the OpenDoc class hierarchy are shown in the following figure. All classes are derived from the superclass ODObject, itself a subclass of somObject, the fundamental SOM superclass (not shown). The figure shows these categories of classes:

- Names in bold represent abstract superclasses that your part editor is likely to subclass.
- Names in italics represent classes whose objects your part typically creates for its own use. You create these objects by calling methods described in Factory Methods.
- Names in plain text represent classes whose objects your part editor calls but typically never has to create; they are created for you by OpenDoc.



Runtime relationships
For an illustrated discussion of the relationships of the principal OpenDoc objects in terms of runtime references among them, see Run-Time Object Relationships.

Classes shown in the following figure are support classes, consisting mostly of iterators and simple sets. They are all direct subclasses of ODObject. (A separate set of specialized OpenDoc support classes, used solely for scripting, is shown in the figure in Scripting-Related OpenDoc Classes).



Compared to some application frameworks, there is little inheritance in the hierarchy represented in the preceding figures; OpenDoc instead makes extensive use of *object delegation* , in which objects unrelated by inheritance cooperate to perform a task. This relatively flat inheritance structure preserves the language-neutral flavor of OpenDoc and improves ease of maintenance and replaceability.

The OpenDoc classes can be divided into these three groups, based on how a part editor might make use of them:

- The class (ODPart) that you must subclass and implement to create your part editor
- The bulk of the implemented OpenDoc classes, whose objects are created either by your part editor or by OpenDoc, that your part editor calls to perform its tasks
- The classes you can subclass and implement to extend OpenDoc

The following sections summarize these groups of classes.

Classes you Must Subclass

The OpenDoc classes listed in this section are abstract superclasses that you are intended to subclass if you wish to implement their capabilities.

The Class ODPart

The class ODPart is central to OpenDoc; it is the one class that you must subclass to create a part editor. ODPart represents the programming interface that your part editor presents to OpenDoc and to other parts.

ODPart is an abstract superclass with approximately 65 defined methods. When you subclass ODPart, you must override all methods, but you need to provide meaningful implementations only for those methods that represent capabilities actually supported by your part editor. The rest can be stub implementations.

There is one additional class that you must subclass and implement if your part is a container part. You must provide an iterator class (a subclass of the abstract superclass ODEmbeddedFramesIterator) to allow callers to access all of the frames directly embedded in your part.

Classes for Extending OpenDoc

OpenDoc provides these classes specifically for enhancing its features:

ODExtension	This class is the abstract superclass from which extensions to OpenDoc are defined; OpenDoc allows you to add new methods to existing classes by associating objects of class ODExtension with the specific class that they extend. The classes ODSemanticInterface and ODSettingsExtension are examples of currently existing subclasses of ODExtension.
ODSemanticInterface	This class, when subclassed, represents the interface through which a part receives semantic events, thus allowing it to be scriptable.
ODSettingsExtension	This class, when subclassed, represents an object with which your part editor can create and display a Settings dialog box for your part editor.
ODViewExtension	This class, when subclassed, represents the interface through which a part can add columns to a details view.
ODDispatchModule	This class is used to dispatch certain types of events (such as keystroke events) to part editors. You can provide for dispatching of new types of events or messages to your part editor by subclassing ODDispatchModule.
ODFocusModule	This class is used to describe a particular type of focus (such as the selection focus). You can provide new types of focus for your part editor by subclassing ODFocusModule.
ODTransform	This class represents a graphical transformation matrix used for drawing. You can subclass it to extend the kinds of transformations it can perform. This class is somewhat different from the other classes in this category, because it can be used as is; for this reason, the class also appears in the next section.

Classes you Can Use

The OpenDoc classes listed in this section implement most of the OpenDoc features that your part editor uses. By using or instantiating objects of these classes, your part can function within an OpenDoc document and can embed other parts.

Abstract Superclasses

These classes are never directly instantiated. The classes ODPart and ODEmbeddedFramesIterator, mentioned in the previous section, are abstract superclasses. Other abstract superclasses are mentioned in Classes for Extending OpenDoc.

The abstract superclasses in the following list are special; they define the basic inheritance architecture of OpenDoc. Not only would you not instantiate these classes, but you would probably never directly subclass them; in developing a part editor, you would use (or further subclass) one of their existing subclasses.

ODObject	This is the abstract superclass for most of the principal OpenDoc classes. All subclasses of ODObject define objects that are extensible.
ODRefCntObject	This is the abstract superclass for reference-counted objects-objects that maintain a count of how many other objects refer to them so that OpenDoc can manage memory use efficiently.
ODPersistentObject	This is the abstract superclass for persistent objects-objects whose state can be stored persistently.

Implemented Classes

These classes are implemented in ways unique to each platform that supports OpenDoc. Some represent objects that are created only by OpenDoc, whereas others represent objects that your part editor may need to create.

The session object	The class **ODSession** represents the user's opening of and accessing to a single OpenDoc document. There is only one ODSession object. Part editors and other OpenDoc objects cache a reference to the single ODSession object for performance reasons. The OpenDoc run time calls ODSession to instantiate all of the OpenDoc library objects (such as ODDispatcher, ODWindowState, and ODArbitrator) and initialize them. ODSession also keeps internal references to these OpenDoc objects, so it is a handy thing to use to access other objects.
The binding object	The class **ODBinding** represents the object that performs the binding of part editors to the parts in a document.
Storage classes	The primary classes of objects associated with document storage are the class **ODStorageSystem** and the set of classes (**ODContainer**, **ODDocument**, **ODDraft**, and **ODStorageUnit**) that constitute a *container suit*e . The container-suite objects all work closely together and are implemented differently for each platform or file system.
Data-interchange classes	The classes **ODLink**, **ODLinkSource**, **ODLinkSpec**, **ODDragAndDrop**, and **ODClipboard** all relate to transfer of data from one location to another. These objects do not represent documents, but they nevertheless use storage units to hold the data they transfer.
Drawing-related classes	The classes **ODCanvas**, **ODShape**, and **ODTransform** represent imaging structures in a given graphics system. The classes ODWindowState and ODWindow represent windows on a given platform. The classes **ODFrame** and **ODFacet** represent the frame and facet structures that define the layout of embedded parts.
Event-handling classes	The classes **ODArbitrator** and **ODDispatcher** control what kinds of user events are sent to which part editors during execution. The classes **ODMenuBar**, **ODPopupMenu**, **ODHelp**, **ODStatusLine**, and **ODUndo** give part editors access to the user interface.
Semantic-event classes	Besides the abstract class **ODSemanticInterface**, the classes **ODNameResolver** and **ODMessageInterface** are associated with sending or receiving semantic events (scripting messages), and connecting to a scripting system. The first figure in A Set of Classes, Not a Framework does not show the entire object hierarchy involved with semantic events and scripting; see the first figure in A Set of Classes, Not a Framework for a more complete picture.

Service Classes

These classes exist mainly as services for other classes to use:

- The **ODStorageUnitCursor** and **ODStorageUnitView** classes facilitate access to specific values in specific storage units.
- The **ODFocusSet** class provides convenient grouping of foci (access to shared resources such as the keyboard or menu bar) for activation and event handling.
- The **ODNameSpaceManager** and **ODNameSpace** classes provide convenient storage for attribute/value pairs. The classes **ODObjectNameSpace** and **ODValueNameSpace**, subclasses of ODNameSpace, provide, respectively, name spaces for OpenDoc objects and for data.
- The **ODTranslation** class provides platform-specific translation between data formats.
- The object descriptor classes (shown in the figure in Scripting-Related OpenDoc Classes) provide scripting support in the form of an object-oriented encapsulation of OSA event descriptor structures.
- Many classes have associated iterator classes and list classes that are used for counting through all related instances of the class.

Writing OpenDoc Software

This section briefly summarizes some high-level aspects of the design and implementation of a part editor. It discusses:

- Issues related to developing with the System Object Model that underlies all OpenDoc classes
- Protocols that your part can participate in to accomplish its tasks
- Several development scenarios for creating OpenDoc software

---

Developing with SOM and IDL

OpenDoc is implemented as a shared library consisting of a set of SOM classes. The interfaces to SOM classes must be written in the SOM Interface Description Language (IDL) and compiled by the SOM compiler, usually separately from the implementations of the classes.

The implementation of OpenDoc objects as SOM objects has several advantages to its use as a shared library:

- SOM objects are dynamically bound. Dynamic binding is essential to the nature of OpenDoc, in which new parts can be added to existing documents at any time.
- All SOM objects, whether their implementations were compiled under different compilers in the same programming language or in different languages, communicate consistently and pass parameters consistently.
- Unlike with other object-oriented architectures, the modification and recompilation of a SOM class do not necessarily require the subsequent recompilation of all of its subclasses and clients.

Because OpenDoc consists of SOM classes, the class ODPart is naturally a SOM class. If you want your part editor, which is a subclass of ODPart plus any other classes that you define, to also consist of SOM classes, then you must write your interfaces in IDL, separate from your implementations, and you must compile them with the SOM compiler. The result of the compilation is a set of header and stub implementation source files, in one of the procedural or object-oriented programming languages supported by the SOM compiler. You complete your development by writing your implementations into the stub implementation files and compiling them, along with the headers, using a standard compiler for your programming language.

If you write your part-editor interfaces in IDL, you will notice that the IDL syntax is very similar to that of C and C++. It includes essentially the same character set, white space rules, comment styles, preprocessing capabilities, identifier-naming rules, and rules for literals. But there are a few differences in source-code appearance to note when declaring or calling methods of SOM-based objects:

- In IDL method declarations (function definitions), each parameter declaration is preceded by a directional attribute ("in", "out", or "inout") that notes whether the parameter is used as an input, or as a result, or as both. (For example, `void FacetAdded (in ODFacet facet);`)
- The C++ interface to any method of a SOM object includes an extra initial parameter, the environment parameter (ev), used by all SOM methods to pass exceptions. See SOM Exception Handling for more information.
- The C interface to any SOM method includes another extra parameter, before the environment parameter, specifying the object to which the method call is directed.

Advantages of making all your classes SOM classes include a greater ability to develop portions of your part editor (or set of editors) using different programming languages and compilers, and a lesser need for recompilation of all of your code when you change portions of it. These advantages may be compelling only if your libraries are very large, however, because they must be balanced against the disadvantages of working in both IDL and a separate programming language.

You are not required to make your part-editor classes SOM classes. If you are developing in C++, for example, you can use C++ classes instead. The generally preferred procedure is to create only one SOM class, a subclass of ODPart whose interface contains nothing but your public methods (overrides of the methods of ODPart and its superclasses). That SOM class delegates all of its method calls to a C++ wrapper class, which contains the functionality for the public methods as well as any private fields and methods. Additional classes can be subclasses of your C++ wrapper class.

Advantages of developing in C++ with a single wrapper object include a lesser need to work with interfaces in two languages (IDL and C++), a smaller memory and calling overhead for your objects, and the availability of C++ features (such as templates) that are not supported by SOM.

SOM Class ID and Editor ID

A *SOM class ID* is an ISO string whose format is *"module::className"*. A part editor's *editor ID* is a SOM class ID that uniquely defines the editor; you need to specify your editor ID in your editor's IDL interfaces. For example, the editor ID for AcmeGraph 1.0 might be "Acme::AcmeGraph". Editor IDs are used for binding; see Information Used for Binding for more information.

For a more detailed description of the Interface Definition Language and instructions on programming with SOM, see, for example, the *SOMObjects Developer Toolkit Users Guide* and *SOMObjects Developer Toolkit Programmers Reference Manual*.

Direct-To-Som (DTS)

SOM defines bindings for C and C++ languages. These bindings consist of a number of macros plus structure or class definitions in header files with the extensions .h and .ih (for C) and .xh and .xih (for C++). They are generated for a particular SOM class by running the SOM compiler on the .idl file for that class interface. The bindings can be used with a wide range of C and C++ compilers and do not require special compiler support.

Direct-To-SOM (DTS) is a new and much more flexible way of using SOM in a C++ program. DTS class definitions resemble regular C++ classes, and you can either write them directly or use the SOM compiler to generate them into .hh files from existing IDL. DTS C++ class definitions can only be used with C++ compilers, such as VisualAge C++, that support DTS.

DTS provides the same access to SOM functionality that the C++ bindings do but, in addition, DTS supports far more of the C++ language. DTS supports:

- Member operators
- Conversion functions
- User-defined **new** and **delete** operators
- Function overloading
- Stack local SOM objects
- First-class source debugging support for SOM classes

You can write and subclass your DTS classes directly and may never need to write any IDL.

For more information about DTS, refer to your compiler programming guide.

OpenDoc Protocols

OpenDoc imposes very few restrictions on how your part editor does its job. The inner workings of your editor's core data engine are of little concern to OpenDoc. The engine should be able to free storage when requested, it should adequately handle cases where the part can be only partially read into memory, and it should handle any multiprocessing or multithreading issues that arise. Other than that, it simply needs to provide an interface to OpenDoc and use the OpenDoc interface to accomplish OpenDoc-related tasks.

The programming interfaces that your part editor uses (and provides) to perform specific tasks are called protocols. The methods of ODPart, for example, participate in approximately 12 protocols, such as part activation and undo. This section briefly describes the protocols; and the next table lists the methods of ODPart that you will have to override to participate in each protocol.

Which Protocols To Participate In

To implement the simplest possible part, you need to participate in only some OpenDoc protocols, and you need to override only some methods of ODPart. As a minimum, your part editor must be able to:

- Draw its part
- Retrieve its part's data
- Handle events sent to its part
- Store its part's data, if it permits editing

Unless it creates extremely simple parts, your part editor must also provide some sort of command interface to the user. It must then be able to:

- Activate its part
- Handle menus, pop-ups, status lines, and help
- Handle windows, dialog boxes, and palettes

If you wish your parts to be able to contain other parts, your part editor must be able to:

- Embed frames and manipulate them
- Create facets for visible frames
- Store frames

Beyond these capabilities, you may want your parts to have additional capabilities, such as drawing themselves asynchronously, providing data transfer capability, supporting scripting, or others. You can add those capabilities by overriding other methods of ODPart.

Overriding the Methods of ODPart

Your fundamental programming task in creating an OpenDoc part editor is to subclass the class ODPart and override its methods. The following list summarizes the OpenDoc protocols that your part editor can use, and lists the sections in this book that describe each protocol more fully. To create full-featured container parts, your editor must support all of these protocols.

The methods of ODPart involved in each protocol are shown in the table that follows this list.

- **Layout.** The layout and imaging protocols together make up the drawing process. Your part uses the layout protocol for frame manipulation and for facet manipulation.

    - Frame manipulation includes adding and removing display frames, setting frame characteristics and order, opening frames into windows, and performing frame negotiation (modifying embedded-frame shapes on request). Your part interacts with its containing part and with its draft object to modify your display frames.

    - Facet manipulation is the adding, altering, or removing of facets of frames that are modified or scrolled into or out of view. You interact with existing facet objects to add or manipulate embedded facets. OpenDoc uses the facet hierarchy constructed by the layout protocol for passing geometric events like mouse clicks to the appropriate part editor.

  All parts that are visible must participate in this protocol. The layout protocol is described, along with the embedding protocol, in Frame and Facet Hierarchies.

- **Imaging.** Your part uses the imaging protocol after the layout protocol, to draw itself appropriately in each of its display frames. Drawing may occur asynchronously, it may occur in response to update events, and it may occur when activation or deactivation affects highlighting. Drawing can be to a video display or to a printer. You interact with frames, facets, canvases, transforms, and shapes during the imaging process.

  All parts that are visible must participate in this protocol. The imaging protocol is described in Canvases, Transforms and Shapes, Drawing, and Printing.

- **Activation.** Through this protocol your part activates and deactivates itself (and its frames and facets). Your part interacts with the arbitrator to change ownership of the selection, menu, and keystroke foci, and to notify the parts and frames involved of the changes.

  Your part must participate in this protocol if it ever needs to be active or if it needs any of the other focus types. The activation protocol is described in Focus Transfer.

- **User events.** OpenDoc uses this protocol to distribute events such as keystrokes and mouse clicks to the appropriate part editors. The document shell, the arbitrator, and the dispatcher use focus-ownership information and the facet hierarchy to deliver these events to your part.

  Your part must participate in this protocol if it handles events. The user-events protocol is introduced in About Event Handling in OpenDoc and is described in more detail in other parts of User Events.

- **Storage.** Your part uses the storage protocol to write its content and its state information persistently in its document, and subsequently to retrieve it from the document. You interact with your draft object and your part's storage unit when reading and writing your part, and you may also interact with other drafts and storage units when using the clipboard, drag-and-drop, and linking protocols.

  All parts must participate in this protocol. The part-storage protocol is described in Storage Units, Properties, and Values and Documents, Drafts, and Parts.

- **Binding.** The OpenDoc document shell controls the runtime binding of your part editor to the parts in a document that it can edit. Your part's methods are not called during binding. The binding protocol is discussed in Binding.

- **Embedding.** Your part uses this protocol to embed other parts within itself and to interact with those parts.

Your part participates in this protocol if it is a container part-that is, if it is capable of embedding parts within itself. The embedding protocol is described in Frame and Facet Hierarchies.

- **Clipboard.** The clipboard protocol allows the user to use menu commands to move content elements and embedded parts into or out of your part, from or to a system-maintained buffer. Your part interacts with the clipboard object both when copying to the clipboard and when pasting from it.

  Your part must participate in this protocol if it supports clipboard transfer. The clipboard protocol is described in Clipboard Transfer.

- **Drag and drop.** The drag-and-drop protocol allows the use of direct manipulation to move content elements and embedded parts into or out of your part or within your part. Your part interacts with the drag-and-drop object.

  Your part must participate in this protocol if it supports drag and drop. The drag-and-drop protocol is described in Drag and Drop.

- **Linking.** Your part can use the linking protocol to export updatable data to another part or to incorporate or embed an updatable copy of another part's data into your part. If your part is the source of a link, you interact with a link-source object; if your part is the destination of a link, you interact with a link object.

  Your part must participate in this protocol if it supports linking. The linking protocol is described in Linking.

- **Undo.** Your part uses this protocol to give the user the capability of reversing the effects of recently executed commands. Your part interacts with the undo object to accomplish this.

  Your part must participate in this protocol if it has an undo capability. The undo protocol is described in Undo.

- **Extensions.** The extension protocol is a very general mechanism for extending your part's capabilities; it consists of an interface in the form of a specialized extension object that other part editors can access.

  Your part participates in this protocol if it provides OpenDoc extensions. The extensions protocol is described in OpenDoc Extension Interface.

- **Semantic events.** Your part uses the semantic events protocol to make itself scriptable. It interacts with the semantic interface object when receiving scripting messages (semantic events); it interacts with the message interface object when sending scripting messages.

  Your part must participate in this protocol if it is scriptable. The semantic events protocol is described in Scripting and OpenDoc.

- **Memory management.** The OpenDoc document shell manages the memory needed for the document containing your part.

  In general, your part editor does not need to be concerned with memory management except to make sure that it deletes or releases objects that it has created or obtained. The memory-management protocol and the use of reference counting is further discussed in Creating and Releasing Objects.

The following table lists the methods of ODPart that you must override to have a functioning part editor, as well as those that you can optionally override to participate in specific protocols. Note that some protocols, such as layout, imaging, and activation, are required of all part editors, and you therefore must override some or all of the methods associated with them. Other protocols, such as embedding or undo, are not required, and you therefore need not override any of their methods if your parts do not participate. It is, of course, strongly recommended that your parts participate in all protocols that are appropriate to their content model.

Protocol	Required overrides	Optional overrides
Layout	AttachSourceFrame	AcquireContainingPartProperties*
	ContainingPartPropertiesUpdated	RevealFrame*
	DisplayFrameAdded	
	DisplayFrameClosed	
	DisplayFrameConnected	
	DisplayFrameRemoved	
	FacetAdded	
	FacetRemoved	
	FrameShapeChanged	
	GeometryChanged	
	Open	
	SequenceChanged	
Imaging	CanvasChanged	AdjustBorderShape*
	Draw	CanvasUpdated*
	GetPrintResolution	
	HighlightChanged	
	PresentationChanged	
	ViewTypeChanged	
Activation	AbortRelinquishFocus	
	BeginRelinquishFocus	
	CommitRelinquishFocus	
	FocusAcquired	
	FocusLost	

User events	AdjustMenus HandleEvent	CreateRootMenuBar
Storage	CloneInto** ClonePartInfo Externalize** ExternalizeKinds InitPart InitPartFromStorage ReadPartInfo WritePartInfo	somInit** somUninit**
Binding	CangeKind	
Memory Management	ReleaseAll**	Acquire** Purge** Release**
Linking	LinkStatusChanged	BreakLink BreakLinkSource CreateLink EditInLinkAttempted* FulfillPromise*** LinkBroken LinkUpdated RevealLink ShowLink UpdateFromLinkSource
Embedding		CreateEmbeddedFramesIterator* EmbeddedFrameUpdated* RemoveEmbeddedFrame* RequestEmbeddedFrame* RequestFrameShape* UsedShapeChanged*
Clipboard		FulfillPromise***
Drag and Drop		DragEnter DragLeave DragWithin Drop DropCompleted FulfillPromise***
Undo		DisposeActionState ReadActionState*** RedoAction UndoAction WriteActionState***
Extensions		AcquireExtension** HasExtension** ReleaseExtension**
Semantic Events		EmbeddedFrameSpec*

* Required of all parts that support embedding
** Defined in a superclass of ODPart
*** Optional even if you implement this protocol

Generally, you must override all of the optional methods listed for a given protocol (other than those marked with *** in the above table) if you are to participate in that protocol. For example, to participate in the extensions protocol, you must override all three methods AcquireExtension, HasExtension, and ReleaseExtension. The embedding protocol has even further requirements; to support embedding, you must not only override all the optional methods listed for that protocol, but you must override several methods associated with other protocols (marked with * in the above table).

Development Scenarios

This section provides a high-level discussion of several possible OpenDoc development scenarios. Reading the scenarios may help you to decide what kinds of OpenDoc component software you are most interested in developing. Specifically, it discusses

- Creating a part editor for noncontainer parts
- Creating a part editor for container parts
- Converting a conventional application into a part editor
- Developing OpenDoc components that are not part editors

Writing an Editor for NonContainer Parts

Writing a part editor that does not support embedding is somewhat simpler than writing an editor that does. Furthermore, if you are starting from scratch (not modifying an existing application), you are free to consider all aspects of the design of your part editor before you implement anything.

- **Content model.** Create a content model that defines the functioning of your part editor and the OpenDoc protocols that is participates in. If your parts are to be scriptable (see the **Scripting** step), your part's content objects and operations must reflect that content model.

- **Core engine.** Design and implement your core data engine, the set of data structures and behaviors that manifest the basic purpose of your editor.

- **Storage.** Implement persistent storage in these situations:

    - Use the OpenDoc storage system to store your part editor's data. Your part editor must implement code to initialize a part and to write it back to storage.

    - Implement clipboard and drag-and-drop capabilities to transfer information between parts, using the same OpenDoc storage concepts for data transfer as for persistent storage.

    - Implement linking support, using a combination of event-handling code and storage code (similar to support for document storage, clipboard, and drag and drop).

- **Drawing.** Give your part the capability of drawing its content, properly transformed and clipped, in whatever combination of facets and frames the content appears, onscreen or offscreen, and for screen display or for printing.

- **Event handling.** Give your part editor the capability of responding to user events such as mouse clicks and keystrokes. It must also respond properly to activation and deactivation events and must use the OpenDoc user interface objects such as members, pop-ups, help, and status line.

- **Scripting.** To support scripting, you must first define the content model of your parts, as noted earlier (in the **Content Model** step). Then you must implement accessory functions to resolve object specifiers (external references to a part's content objects) as well as semantic-event handlers to perform your part's content operations.

- **Extensions.** If you plan to extend the capabilities of your parts to communicate with other parts or process information fast, create and attach an extension interface to your part editor. To obtain existing public extension interface designs or to propose a new public interface, contact CI Labs.

- **Packaging and shipping.** Once your part editor is complete, provide information for the OpenDoc binding process to use, indicating what part kinds are handled and what semantic events are handled.

You ship your product as one or more part editors, plus an install program and accompanying documentation. Users install your part editor into their systems and then, using the templates that get control at part registration time, create new documents of your part kind or insert new parts with your part kind into their documents. Rules and conventions for installing part editors and storing documents vary among platforms. For OS/2 platform rules, see the recipe Installation of OpenDoc Software.

Users themselves can create additional, customized template documents. Users should be able to exchange documents, including templates, freely.

Writing an Editor for Container Parts

If your part editor is to support embedding-that is, if its parts are to be container parts-you need to include the following additional steps:

- **Embedding.** You need to add embedding support to your content model and to storage.

    - Your content model needs to include a type of content element that represents an embedded part. If your part editor supports semantic events, embedded parts must be content objects to which you can pass events.

    - Make sure your parts can store embedded parts, both in their documents and during data transfer. This is relatively

simple to implement; OpenDoc takes care of most of it.

- **Layout management.** You need to add support for layout management during event handling and during frame negotiation.
    - Your part editor must be able to maintain updated information on embedded frames and facets and notify embedded parts of such changes. It must add facets when embedded frames become visible. It must modify or delete facets when embedded frames move or become no longer visible.
    - Your part editor must include support for layout negotiation, including updating the shapes and transforms associated with each visible embedded frame and facet.

For a summary of the issues to consider in creating a part editor for container parts, see Embedding Checklist.

Converting a Conventional Application to a Part Editor

Creating a part editor from an existing conventional application involves maintaining its core features but repackaging them for the OpenDoc environment.

- **Content model and core data engine.** You should have your content model and core data engine already in place. You may need to separate your core data engine from other facilities, such as user interface and storage, if it is not already sufficiently separated.
- **Storage.** You must refit all file I/O and clipboard data transfers into OpenDoc terms, as described for part editors in the **Storage** step in Writing an Editor for NonContainer Parts.
- **Event handling.** Because your part editor will receive its event information from the document shell, you need to remove your event loop and event-handling code.
- **Scripting.** You can add scripting support, as described for part editors in the **Scripting** step in Writing an Editor for NonContainer Parts.
- **Extensions.** You can extend the capabilities of your parts, as described for part editors in the **Extensions** step in Writing an Editor for NonContainer Parts.
- **Packaging and shipping.** Package and ship your part editor, as described for part editors in the **Packaging and Shipping** step in Writing an Editor for NonContainer Parts.

If your converted application is to be a container part, you need to follow these additional steps:

- **Embedding.** Add embedding support, as described for part editors in the **Embedding** step in Writing an Editor for Container Parts.
- **Layout management.** Add layout-management code, as described for part editors in the **Layout Management** step in Writing an Editor for Container Parts.

Writing Other Types of Component Software

This book primarily describes how to create a part editor. However, you are not limited to that alone. Development of container applications is discussed briefly in the previous section. Using the OpenDoc class libraries, you can also create other kinds of OpenDoc component software:

- **Part viewers** are simply part editors with their editing and saving capabilities removed. A part editor should be able to read, display, and print its parts.

  Development issues for part viewers are a subset of those for fully functional part editors. A part viewer is usually much smaller than its corresponding editor, and it can be developed from the same code base.

  To encourage data interchange, you should always create a part viewer for every part editor you develop, and you should distribute the viewer widely and without cost or obligation to the user.
- **Part extensions** are objects with which you can extend the programming interface of your part editor. Extensions work in conjunction with part editors, allowing their parts to communicate with and directly manipulate other parts. Extensions are described in Extending OpenDoc.
- **Document-shell plug-ins** are extensions to the capabilities of the shell. They are DLLs that you can write and install. Shell plug-ins are described in Shell Plug-Ins.

- **Part or document services** are OpenDoc components that, instead of editing and saving parts in a document, provide specialized services to those parts and documents. Spelling checkers, database-retrieval engines, and network connection services are all examples.

  You develop an OpenDoc service much as you develop a part editor. Like part editors, services are subclasses of ODPart. However, services commonly use less of the embedding, layout, and imaging protocols of OpenDoc, and they usually communicate with the parts they serve through an extension interface (a subclass of ODExtension). The extension interface is described in Extending OpenDoc.

Frames and Facets

This is the first of eight chapters that discuss the OpenDoc programming interface in detail. This chapter focuses on the key concepts of how parts use frames and facets to accomplish embedding and to communicate with each other during layout and display.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. For additional concepts related to your part editor's runtime environment see OpenDoc Run-Time Features.

This chapter starts with a general discussion of frames and facets, and then describes:

- How your part uses its display frames and facets to function as an embedded part in an OpenDoc document
- How your part can negotiate with its containing part for modifications to its display frames

If your part is a noncontainer part, these are the only sections of this chapter you need to read. If, however, you are developing a container part, you also need to read the remainder of this chapter. It describes:

- How to negotiate with your embedded parts for modifications to their display frames.
- How to otherwise manipulate the frames and facets of your embedded parts

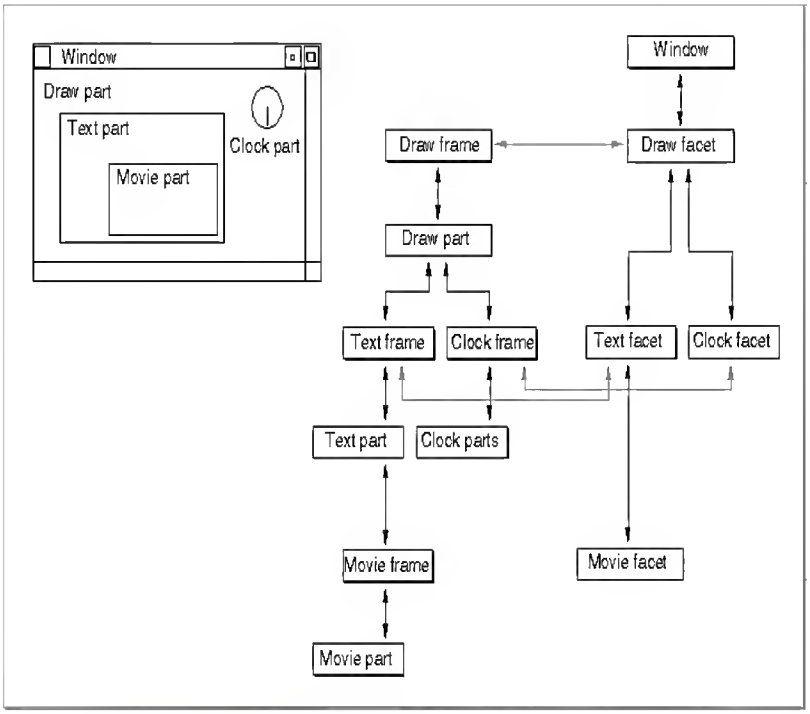
For a general summary of the embedding process, see Adding an Embedded Part. For a summary of embedding capabilities that must be implemented by container parts, see also Embedding Checklist.

Frame and Facet Hierarchies

The object hierarchy of embedding controls how information is passed among the parts that make up a compound document. Parts and embedded frames make up one hierarchy; facets make up a separate but essentially parallel hierarchy.

The following figure shows a simple example of these hierarchies for a document that consists of a graphics root part ("draw part"), in which is embedded a clock ("clock part") and a text-processing part ("text part"). The text part has a movie-viewing part ("movie part") embedded within it.

This figure uses the same conventions as the more detailed run-time diagrams presented in the figure in Run-Time Object Relationships. Individual OpenDoc objects are represented by labeled boxes, with the references (pointers in C++) between them drawn as arrows. (Different arrows have different appearances in the following figure for clarity only; all represent the same kinds of object references). The embedding structure extends downward, with more deeply embedded objects shown lower in the diagram.



The window object is at the top of the previous figure, uniting the two hierarchies. The window object itself is referenced by the window state object, as shown in the figure in section Window State and Windows. (Some inter-object references have been omitted from this diagram, as noted in the next section).

Frames and Parts

The fundamental structure of embedding in the compound document is represented by the hierarchy on the left. The draw part is the root part of the document; it directly references its two embedded frames. Those frames, in turn, reference their parts: the text part and the clock part. The text part references its one embedded frame, which in turn references its part, the movie part. Each part thus references its own embedded parts only indirectly. (In this case, there is one frame per part, but there could be more than one).

Note also that each part also has a reference back up the hierarchy to its display frame, and each frame has a direct reference up to its containing frame. See the figure in Embedding for a more complete picture of the object structure of embedding.

The embedding relation of parts and frames does not have to be the strict hierarchy shown in the figure in Frame and Facet Hierarchies. For example, a single part can have frames embedded at different levels, as in the figure in section Providing Split-Frame Views.

Parts and frames are stored together in their document. When the document is closed, the states of all the parts and frames are saved. When the document is reopened, all the parts and frames are restored by reading in the saved data.

Facets

The hierarchy on the right side of the figure in section Frame and Facet Hierarchies is analogous to the one on the left, but it is simpler and more direct. The facet hierarchy is designed for fast drawing and event-dispatching. Each facet corresponds to its equivalent frame, but it directly references its embedded facets. (In this case, there is one facet per visible frame, but there could be more than one).

Whereas frames must exist (at least in storage, if not always in memory) for all embedded parts in a document, facets are needed for only those frames that are visible at any one moment. Because the facet hierarchy is for drawing and event dispatching, there is no need to have facets that cannot be drawn and cannot accept events.

Facets are not stored when a document is saved. Facets are created and deleted by their frames' containing parts, as the facets' frames become visible or invisible because of scrolling, resizing of frames or windows, or opening and closing of windows.

Note from the figure in section Frame and Facet Hierarchies that each frame has a direct reference to (and from) its equivalent facet. That frame-to-facet reference is the connection between the two hierarchies. It also means that each part object references its own facets only indirectly, through its display frames. (Your part can, of course, keep its own private list of its facets).

Working with your Display Frames and Facets

Your part's *display frames* are the frames in which you draw your part's contents. This section discusses how to request or respond to changes and additions to those frames.

You do not directly create your own part's display frames; your part's containing part does that. When your part is first created or first read into memory from storage, the containing part of your part will in most cases have already provided at least one frame for your part to display itself in. However, it is possible that your part can be instantiated without already having a display frame. (see Adding an Embedded Part). Your part should allow for that possibility.

Your part should also allow for the possibility of multiple display frames. Your part must keep a list of its display frames, with enough information so that your part knows how to display itself in each and synchronize their displays when necessary. OpenDoc does not specify the format of that list. It is your responsibility to keep the list and to update each display frame that has been affected by any change to your part.

Responding to Reconnected and Closed Display Frames

When the document containing your part opens and your previously stored display frames are read in and instantiated, OpenDoc calls your part's DisplayFrameConnected method. Here is its interface (note that this and all other method prototypes in this book are written in IDL):

```
void DisplayFrameConnected(in ODFrame frame);
```

Your DisplayFrameConnected method should update your part's internal list of display frames and other related structures to reflect the addition of the frame. It should assign the frame's used shape, if different from the frame shape itself. The method should also check the frame's *content extent* and update it if necessary, as described in Content Extent. The method should also call the frame's SetDroppable method if you wish the frame to be able to accept drops.

Unlike for DisplayFrameAdded (see Responding to an Added Display Frame), your DisplayFrameConnected method does not normally have to create new part info data or a new set of embedded frames for the display frame, because they will have been created earlier.

When your part's document is closed, OpenDoc calls your part's DisplayFrameClosed method when it closes your display frame:

```
void DisplayFrameClosed(in ODFrame frame);
```

Your DisplayFrameClosed method should

1. Update your part's internal list of display frames and other related structures to reflect the closing of the frame
2. Relinquish any foci owned by the frame (see Relinquishing Foci)
3. Call the Close method of all your embedded frames

If this frame is the root frame of your window and you have previously instructed OpenDoc not to dispose of the platform-specific window structure, you must dispose of the platform window yourself at this time.

Closing and reconnecting can happen at other times as well:

- When a part's editor is changed at run-time, OpenDoc calls the DisplayFrameClosed method of the editor losing the part and the DisplayFrameConnected method of the editor receiving the input.
- For efficient memory use, a containing part may not always keep frame objects in memory for all of its embedded frames. Instead, as described in Reconnecting and Releasing Embedded Frames, it might release frames as they become invisible through scrolling, and recreate them as they become visible. Released frames can then be closed by OpenDoc.

Do not update your persistently stored display frames when your DisplayFrameConnected or DisplayFrameClosed methods are called; these methods should have no effect on stored data. In general, you should write your part content and frames to storage only when your Externalize or ExternalizeKinds methods are called.

Responding to Added or Removed Facets

When your part's containing part has added a facet to one of your part's display frames, the display frame notifies your part of the addition by calling your part's FacetAdded method. Here is its interface:

```
void FacetAdded(in ODFacet facet);
```

Your FacetAdded method must perform certain actions to handle the addition of the new facet to one of your frames. Some actions depend on the nature and implementation of your part itself, but others are standard:

- The method should store any private information that it needs as part info data in the facet that is being added. Although a facet's part info is not persistent, it can hold information that the facet needs for display.
- The method should assign an active shape to the facet, if needed. If you do not explicitly assign an active shape, OpenDoc uses the frame shape of the facet's associated frame for the active shape.
- The method should create facets for all embedded frames that are visible within the area of the added facet. See Adding a Facet.

When a containing part removes a facet from one of your part's display frames, the frame notifies your part by calling your part's FacetRemoved method:

```
void FacetRemoved(in ODFacet facet);
```

Your FacetRemoved method must perform certain actions to handle the removal of the facet. In general, this method reverses the actions performed by FacetAdded. Typically, the method should at least:

- Remove the facets for all embedded frames that were visible in the area of the removed facet
- Delete any part info data that was referenced in the facet

Facets are intended to be ephemeral objects; don't hold references to them when they are no longer needed. Your FacetRemoved method should delete all references to the facets that it removes.

Note: If your part is the root part in a window, it receives a FacetAdded call from the root frame when the window is opened and a FacetRemoved call from the root frame when the window is closed.

Resizing a Display Frame

Because of editing operations in your part, or because of an undesirable frame size imposed by your part's containing part, you may wish to change the size or shape of your display frame. You must negotiate this change with the containing part. (For an example of frame negotiation, showing the point of view of both the containing part and the embedded part, see Frame Negotiation).

You start by requesting a new frame size from your part's containing part. Depending on its current contents and other constraints such as gridding, the containing part may grant the requested size, return a different size, or refuse the request by returning a size identical to your current frame size.

To request a new frame size, take these steps:

- Call the RequestFrameShape method of your display frame. The frame, in turn, calls the containing part's RequestFrameShape method to forward the request.
- The containing part may honor the request, or it may decide on a different (usually smaller) shape. It returns the shape it will let your display frame have. The frame stores it as its frame shape and returns it to you.
- Use the returned frame shape to update the used shape for that frame. At this time, you can also update the active shape for your frame's facet.
- If your part does not wish to accept the new shape, it can call the frame's RequestFrameShape method again, but with a different shape to avoid endless repetition of these steps. Alternatively, it can request an additional frame, as described in Requesting an Additional Display Frame (next).

Requesting an Additional Display Frame

Your part may need a display frame in addition to the frame or frames already created by your part's containing part. For example, you may need an extra frame to flow content into another frame (as when you add another column or page of text), or you may need another frame to display your part with a new presentation.

To request another frame, you can call the RequestEmbeddedFrame method of the containing part. You must specify one of your current display frames as the base frame, the frame that defines certain characteristics of the new frame.

- The new frame is a *sibling* of the base frame; that is, it is embedded at the same level as the base frame.
- The new frame is in the same group as the base frame.
- The new frame has a sequence number within the base frame's frame group, assigned by the containing part. (By convention, the containing part adds the new frame to the end of the sequence in that group.)

You also pass additional information, such as the new frame's view type, presentation, and overlay status (that is, whether it should *overlay* , or float over, the content of the containing part).

The shape you request for the new frame is understood to be expressed in the frame coordinates of the base frame (see Frame Coordinate Space). Thus you can request a position for the new frame that is relative to the base frame (such as above, below, to the side, or overlapping), by specifying an origin that is offset from the base frame's origin. The containing part has ultimate control over frame positioning, however, and is not required to size or place the new frame exactly as you request. Furthermore, the frame shape returned by the containing part is by convention *normalized* ; that is, the relative-positioning information has been stripped from it and its origin is at (0, 0).

The containing part responds to this call as described in Adding an Embedded Frame on Request. Your part then responds as described in Responding to an Added Display Frame (next).

Responding to an Added Display Frame

When an additional display frame is created, OpenDoc automatically connects it to the part it displays. This automatic connection ensures that frames are always valid and usable; the object that creates the new frame need do nothing beyond creating the frame itself.

To achieve that automatic connection, the part displayed in the frame must respond to this method call, which informs the part that it has a new frame:

```
void DisplayFrameAdded(in ODFrame frame);
```

Your part receives this call when it is first created and has no previously stored display frame, and also when additional display frames are created for it. In response to this call, your part's DisplayFrameAdded method performs the appropriate tasks. Most of them depend on the nature and implementation of your part itself; however, here are some general actions it should take:

- The method should add the new display frame to your part's list of display frames. This list, like other internal structures, is completely hidden from OpenDoc. You can represent the list any way you choose.
- The method should validate the view type and presentation of the new frame. Your part should accept any view types that are in the required set of view types, plus any other view types or presentations that you support. The DisplayFrameAdded method should correct these values, if necessary, as described in Defining General Display Characteristics.
- The method should assign your part's current *content extent* to the frame using the frame's ChangeContentExtent method; see Content Extent.
- The method should assign a used shape for the frame. If you do not specifically assign a used shape, the frame shape is used; a containing part that calls the frame's AcquireUsedShape method receives the frame shape in that case.
- The method should add any needed part info to the frame, by calling the frame's SetPartInfo method. See Part Info for more information.
- If the frame being added is the root frame of its window, your part may want to activate itself. Part activation is described in How Part Activation Happens. (This situation can occur only when your part is first opened into a window, such as a part window; it cannot happen when an additional display frame for the current window is created.
- If the frame being added can accept dropped data, the method should call the frame's SetDroppable method, passing it a value of kODTrue. Otherwise, the frame will not be able to receive data through drag and drop. See Drag and Drop for more information.

Your part should not perform any layout or imaging tasks as a result of a display frame being added; specifically, it should not at this point negotiate with its containing part for a different frame shape. It should wait until its FacetAdded method is called, by which time the containing part has stored the frame shape and the frame has become visible.

Note: OpenDoc calls DisplayFrameAdded only when a frame is newly created. When your part opens and its stored display frame is recreated, OpenDoc calls your part's DisplayFrameConnected method; see Responding to Reconnected and Closed Display Frames.

Removing a Display Frame

To remove a frame in which your part is displayed, you call the RemoveEmbeddedFrame method of your part's containing part (see Removing an Embedded Frame). You can remove only those frames that you have previously requested through calls to your containing part's RequestEmbeddedFrame method. Your other display frames can be removed only by your containing part.

When the containing part receives the RemoveEmbeddedFrame call, it permanently severs all connection between your part and the frame (and the draft object may delete the frame from storage if its reference count becomes 0).

Responding to a Removed Display Frame

Your part's containing part can also initiate removal of one of your display frames, as described in Removing an Embedded Frame. The removal may occur, for example, if your frame is part of a selection that is cut or deleted. Regardless of whether your part or the containing part initiates the removal, OpenDoc calls your part's DisplayFrameRemoved method.

This is its interface:

```
void DisplayFrameRemoved(in ODFrame frame);
```

In your DisplayFrameRemoved method, take steps such as these:

1. Relinquish any foci owned by the frame (see Relinquishing Foci).
2. Delete any part info that your part had associated with the frame. Set the frame's part info data to null by calling the frame's SetPartInfo method.
3. Update your part's internal list of display frames and other related structures to reflect the removal of the frame.
4. Remove in turn any embedded frames that your part had been displaying in the removed frame. Check the identity of the embedded frame's containing frame to determine which call to make:
    - If your (removed) display frame is designated as the embedded frame' containing frame, the display frame has not been moved from you part to another object. In this case, call the embedded frame's Remove method.
    - If your (removed) display frame is *not* designated as the embedded frame's containing frame, the display frame has been moved from your part to another object. In this case, call the embedded frame's Release method.

If this frame is the root frame of your window and you have previously instructed OpenDoc not to dispose of the platform-specific window structure, you must dispose of the platform window yourself at this time.

Note: OpenDoc calls DisplayFrameRemoved only when a frame is permanently removed from your part. When your part closes and OpenDoc stores its display frame, OpenDoc calls your part's DisplayFrameClosed method; see Responding to Reconnected and Closed Display Frames.

Grouping Display Frames

As an embedded part, your part does not directly control the frames in which it is displayed. However, if you need to flow text or other content in order through a sequence of separate display frames of your part (as for page-layout purposes). If so, you can request that the containing part create and assign you the needed frames as a sequence in a single frame group. See Creating Frame Groups. If your part's containing part adds a new frame to your display frame's group or reorders the sequence of one of your display frames within its group (by calling the frame's ChangeSequenceNumber method), OpenDoc calls your part's SequenceChanged method. This is its interface:

```
void SequenceChanged(in ODFrame frame);
```

Your part can then retrieve the new sequence number of the frame by calling its GetSequenceNumber method.

Synchronizing Display Frames

Sometimes, views of your part in two or more separate frames must be *synchronized* , meaning that any editing or other changes to the contents of one (the *source frame*) must force updating of the other. In some cases, such as when you open one of your display frames into a window, you can determine these dependencies internally. In other cases, you cannot. For example, if your part is an embedded part and your containing part creates multiple views of your part, your containing part asks you to synchronize those views by calling your part's AttachSourceFrame method. This is its interface:

```
void AttachSourceFrame(in ODFrame frame,
                       in ODFrame sourceFrame);
```

Your AttachSourceFrame method should take whatever action is necessary to synchronize the frames; make sure that the result of any editing in one frame appears in the other. At a minimum, if the two frames have the same presentation, the method should duplicate all embedded frames in one frame into the other (and attach them to their source frames as well). It is the containing frame that determines when embedded frames must be synchronized. See Synchronizing Embedded Frames.

Adopting Container Properties

As described in Transmitting your Container Properties to Embedded Parts, a containing part can notify its embedded parts of the c*ontainer properties* , the characteristics of its content that it expects the embedded parts to adopt. For example, if your part is a text part and the user embeds it into another text part, the containing part may expect your part to adopt the general text appearance (font, size, stylistic variation, and so on) of the containing part.

Your part can, of course, adopt only those container properties whose format and meaning it understands. You obtain the set of container properties that your containing part makes available for adoption by calling the AcquireContainingPartProperties method of your containing part.

A containing part calls your part's ContainingPartPropertiesUpdated method when it changes any of the container properties available for your part to adopt. This is its interface:

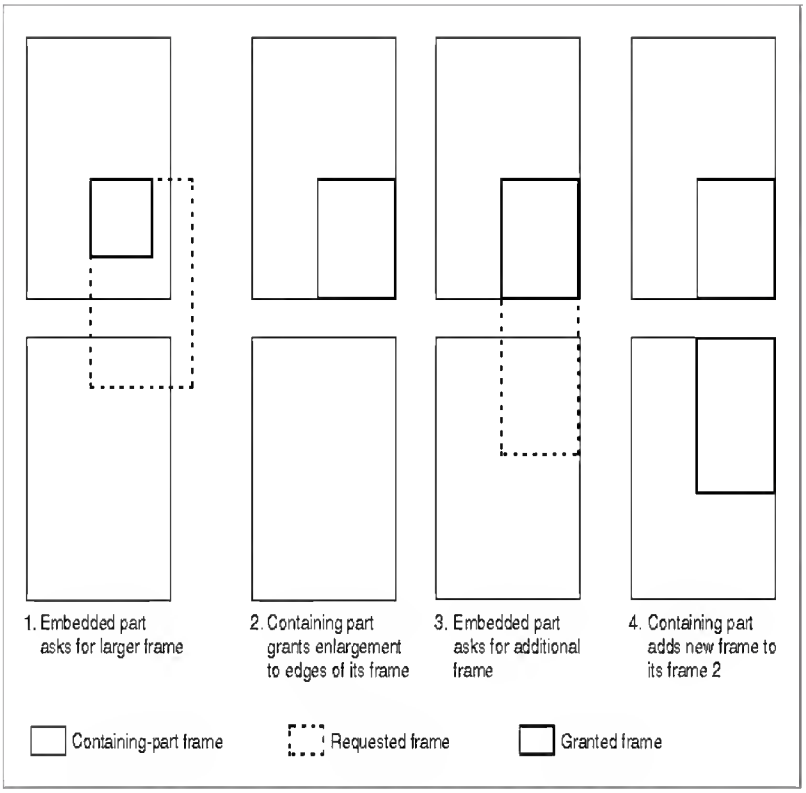
```
void ContainingPartPropertiesUpdated(in ODFrame frame,
                                     in ODStorageUnit propertyUnit);
```

Your ContainingPartPropertiesUpdated method should read and adopt any container properties that it understands from the provided storage unit. Then, it should in turn call the ContainingPartPropertiesUpdated method of any of your part's own embedded frames (other than bundled frames) that are displayed within the frame passed to ContainingPartPropertiesUpdated.

Frame Negotiation

Each part in an OpenDoc document controls the positions, sizes, and shapes of its embedded frames. At the same time, embedded parts may need to change the sizes, shapes, or numbers of the frames in which they are displayed. Read this section if your part expects to negotiate its display frame sizes with its containing part, or if it is a container part and expects to negotiate the sizes of its embedded frames with their parts.

Either party can initiate the negotiation, although the containing part has unilateral control over the outcome. The following figure shows an example of frame negotiation, in which an embedded part with a single display frame requests a larger frame size from its containing part, which has two display frames.



In this case, the embedded part initiates the frame negotiation. Its frame is wholly contained within frame 1 (the upper frame) of the containing part:

1. The embedded part asks the containing part for a significantly larger frame, perhaps to fit material pasted in from the clipboard. The embedded frame is not concerned with, and does not even know, where or how the larger frame will fit into the containing part's content.

2. The containing part decides, on the basis of its own content model, that the requested frame is too large to fit within frame 1. The containing part instead increases the size of the embedded frame as much as it can, assigns it a place in its content area, and returns the resulting frame to the embedded part.

3. The embedded part accepts the frame given to it. If it were to repeat the first step and ask for the original larger frame again, the containing part would simply repeat the second step and return the same frame.

   But the embedded part still wants more area for its display, so it tries a different approach; it requests another display frame, this time to be embedded in frame 2 of the embedded part.

4. The containing part decides that the requested frame will fit in frame 2. It assigns the frame a place within frame 2 and returns the frame to the embedded part.

Frame negotiation, from the point of view of the embedded part, is discussed in Resizing a Display Frame and Requesting an Additional Display Frame. Frame negotiation, from the point of view of the containing part, is discussed in Resizing an Embedded Frame and Adding an Embedded Frame on Request.

Working with Embedded Frames and Facets

Read the information in this section if your part can contain embedded parts. It discusses what information your part needs to maintain for embedded frames and how it creates those embedded frames and facets.

As a containing part, your part needs to maintain current information on the shapes and transforms for all its visible embedded frames and facets. If it makes changes to them, it should not only update its own information but in some cases also notify the embedded parts of the changes, so that they can update their own information. Your part must also support frame negotiation (see Frame Negotiation), to permit embedded parts to request additional frames or changes to the sizes of their existing frames.

If your part is a container part, the user can embed other parts into your part in several ways. Examples include pasting from the clipboard, using drag and drop, or even selecting a tool from a palette.

The overall process of embedding a part is summarized in the section Adding an Embedded Part. The overall process of removing a part is summarized in Removing an Embedded Part. Both processes make use of the specific tasks described in this section.

Providing an Embedded-Frames Iterator

Your part must keep a list of all its embedded frames. OpenDoc does not specify the format of this list. However, your part must implement an iterator class (a subclass of ODEmbeddedFramesIterator) that gives a caller access to each of the frames in your list of embedded frames. The caller creates an iterator to access your embedded frames by calling your part's CreateEmbeddedFramesIterator method, which has this interface:

```
ODEmbeddedFramesIterator CreateEmbeddedFramesIterator(in ODFrame frame);
```

Your implementation of CreateEmbeddedFramesIterator must provide First, Next, and IsNotComplete methods, as do other OpenDoc iterators. See Accessing Objects through Iterators for additional discussion.

Creating a New Embedded Frame

If your part embeds a part that does not already have its own display frame, you need to create a new embedded frame that will be the embedded part's display frame. Also, if you create an additional view of an existing embedded part you need to create and embed a frame to hold the new view.

In these situations, your part-the containing part-initiates the embedding creation. Your method to create an embedded frame calls the CreateFrame method of the draft, which returns an initialized frame that has already been assigned to the embedded part (CreateFrame calls the DisplayFrameAdded method of the embedded part for this purpose). This calling sequence ensures that the new frame can be used as soon as the draft returns it.

When you call CreateFrame, you specify several features of the new frame, including the following:

- Containing frame: the frame (your display frame) that is to contain the embedded frame.
- Frame shape: the shape of the frame to be created. Your part can use a default shape or it can use information supplied with the data you are embedding; see Frame Shape or Frame Annotation for more information.
- Part: the part (your part) in which the frame is to be embedded.
- View type and presentation: the initial view type and presentation the part displayed in the embedded frame is to have.
- Subframe: whether or not the embedded frame is to be a subframe of its containing frame (your display frame). See Using Multiple Facets for examples.
- Overlay status: whether or not the frame should overlay, or float over, the containing frame.

Once the reference to the new frame is returned, take these steps:

1. Store it somewhere in your content and add it to your part's list of embedded frames. It's up to you to decide how your part internally stores its content, including its embedded frames.
2. Set the embedded frame's link status appropriately. See Frame Link Status for more information. If your part does not support linking, you still must set the new frame's link status (to kODNotInLink).
3. If the embedded frame is visible within the containing frame, you must create a facet for it. See Adding a Facet.

Creating a frame should be an undoable action. See Undo and Embedded Frames for information on how to set a frame's in-limbo flag when you undo or redo its creation.

For more information on the DisplayFrameAdded method, see Requesting an Additional Display Frame.

Adding an Embedded Frame on Request

As a container part, your part may also need to support creation of an embedded frame when requested to do so by another part. As described in Requesting an Additional Display Frame, an embedded part can call your part's RequestEmbeddedFrame method in order to get an additional frame for its content. The embedded part passes the necessary information about the requested frame, as shown in this

interface:

```
ODFrame RequestEmbeddedFrame(in ODFrame containingFrame,
                             in ODFrame baseFrame,
                             in ODShape frameShape,
                             in ODPart embedPart,
                             in ODTypeToken viewType,
                             in ODTypeToken presentation,
                             in ODBoolean isOverlaid);
```

Your RequestEmbeddedFrame method passes most of this information along to your draft's CreateFrame method. The base-frame parameter specifies which of the embedded part's existing display frames is to be the base frame for the new frame; the new frame will be a sibling of the base frame and will be in the same frame group (if any) as the base frame.

The frameShape parameter passed to this method expresses the requested frame shape in the coordinate system of the base frame. By this method, the embedded part can request, by specifying the origin of the frame shape, a relative location for the new frame compared to its base frame. Your RequestEmbeddedFrame method should take this information into account when granting the frame shape and assigning an external transform to its facet. Specifically, you should incorporate the positioning information into the external transform (if appropriate, given the nature and state of your intrinsic content) and then return a frame shape that has been normalized-that is, one in which the origin of the frame shape is at (0, 0).

Based on information in the existing frame, your RequestEmbeddedFrame method should also assign the new frame's group ID and sequence number, by calling its SetFrameGroup and ChangeSequenceNumber methods. Your part can assign the new frame any sequence number in the frame group, although by convention you should add the new frame to the end of the current sequence. Then the RequestEmbeddedFrame method should add the new frame to your part's list of embedded frames. The method should also create a facet for the new frame if it is visible.

Other tasks you perform are the same as when you initiate the creation of the frame (see Creating a New Embedded Frame). You might implement your RequestEmbeddedFrame method in such a way that it calls a private method to actually create the frame. You could then use that same private method in both situations.

Resizing an Embedded Frame

To change a frame's size, the user typically selects the frame and manipulates the frame border's resize handles. The containing part is responsible for drawing the selected frame border, determining what resize handles are appropriate, and interpreting drag actions on them.

You need to notify an embedded part that its frame has changed in these situations:

- If your part is the containing part of a frame that is resized by the user
- If your part has other reasons to change the size of an embedded frame (for example, to enforce gridding or in response to editing of your own intrinsic content surrounding the frame)

Return the changed shape (the same shape reference passed to you) in the method result. After you change the embedded frame's shape, call the frame's ChangeFrameShape method and pass it the new shape. (For efficiency, you can first acquire the embedded frame's existing frame shape, then resize it, then call ChangeFrameShape, and finally release your reference to the frame). The frame in turn notifies its part by calling its FrameShapeChanged method. In response, the embedded part may request a different frame size, as discussed in Resizing a Display Frame.

Note that resizing may result in your part having to adjust the layout of its own intrinsic content, such as wrapped text.

Removing an Embedded Frame

You may need to remove an embedded frame from your part, either as a direct result of user editing (such as cutting or clearing a selection) or upon request of the part displayed in that frame.

An embedded part requests that you remove one of its display frames by calling your part's RemoveEmbeddedFrame method. This is its interface:

```
void RemoveEmbeddedFrame(in ODFrame embeddedFrame);
```

Whether because of editing or by request to RemoveEmbeddedFrame, the basic procedure for removing an embedded frame is to delete all

its facets, delete it from your private content structures, and call its Remove or Release method. OpenDoc then notifies the embedded part of the removal by calling its DisplayFrameRemoved method, as described in Removing a Display Frame.

Removing an embedded frame should be an undoable action. Therefore, you should add a few steps to the procedure to retain enough information to reconstruct the frame (and all its embedded frames) if the user chooses to undo the deletion. You can follow steps such as these:

1. Remove all of the embedded facets from the frame; see Removing a Facet.
2. Set the frame's containing frame to kODNULL, to indicate that the frame is no longer embedded in any part.
3. Place a reference to the frame in an undo action that you add to the undo history; see Adding an Action to the Undo History.
4. Remove the frame from your embedded-frames list (with which you allow callers to iterate through your embedded frames). Set the removed frame's in-limbo flag to true, as shown in the table *Setting a Frame's In-Limbo Flag* in Undo and Embedded Frames.
5. Remove the frame from your other private content structures (except for your undo structures).

If the user subsequently chooses to undo the action that led to the removal of the frame, you can then

1. Retrieve the reference to the frame from the undo action, and set its in-limbo flag to false, as shown in the table *Setting a Frame's In-Limbo Flag* in Undo and Embedded Frames.
2. Reestablish your display frame as the frame's containing frame
3. Recreate any needed facets for the frame

If the undo action history is cleared, your part's DisposeActionState method is called and at that point you can remove the frame object referenced in your undo action. You call the frame's Remove method if its in-limbo flag is true; you call the frame's Release method if its in-limbo flag is false.

Reconnecting and Releasing Embedded Frames

Your part connects and closes its embedded frames when the document containing your part opens and closes. On opening, when your part calls the draft's AcquireFrame method for each embedded frame that you want to display the frame in turn calls the DisplayFrameConnected method of its part. After your part and its embedded frames have been saved, and before the document closes, you call the Close method of each of your embedded frames; the frames in turn call the DisplayFrameClosed methods of their parts.

The DisplayFrameClosed and DisplayFrameConnected methods are described in Responding to Reconnected and Closed Display Frames.

For efficient memory usage, your part can retrieve and connect only the embedded frames that are visible when its document opens, and it can also release and reconnect embedded frames during execution, as the frames become invisible or visible through scrolling or removal of obscuring content. This process is described in Lazy Instantiation.

Adding a Facet

OpenDoc needs to know what parts are visible in a window and where, so that it can dispatch events to them properly and make sure they display themselves. But OpenDoc does not need to know the embedding structure of a document; that is, OpenDoc is not directly concerned with where embedded frames are located and what their sizes and shapes are. Because embedded frames are considered to be content elements of their containing part, each containing part maintains, in its own internal data structures, embedded-frame positions, sizes, and shapes. Therefore, containing parts need to give OpenDoc information about embedded frames only when they are visible. They do this is by creating a facet for each location where one of their embedded frames is visible.

An embedded frame in your part may become visible immediately after it is created, or when your part has scrolled or moved it into view, or when an obscuring piece of content has been removed. No matter how the frame becomes visible, your part must ensure that there is a facet to display the frame's contents. Follow these general steps:

1. If your own display frame has multiple facets, create an embedded facet for each of the facets in which the newly visible embedded frame appears. To do that, iterate through all the facets of your display frame, creating embedded facets where needed.
2. For each needed facet, if it does not already exist, make one by asking the containing facet (your display frame's facet) to create one. Call the containing facet's CreateEmbeddedFacet method.
    - In your call to CreateEmbeddedFacet, assign the embedded facet a clip shape equal to the embedded frame's frame shape, if the new facet is to be positioned in front of all sibling facets and other content in your part. Otherwise, adjust

the clip shape as described in Managing Facet Clip Shapes.

- Assign the embedded facet an external transform, based on your part's internal data on the location of the embedded frame.
- If you will draw offscreen through this facet, assign the facet a canvas. See Canvases for an explanation.

3. If you want to receive events sent to, but not handled by, the part displayed in this facet, set the event-propagating flag of the facet's frame. See Propagating Events for an explanation.

After you create the facet, OpenDoc calls the embedded part's FacetAdded method to notify it that it has a new facet.

Removing a Facet

An embedded frame may become invisible when the containing part has deleted it, scrolled or moved it out of view, or placed an obscuring piece of content over it. In any of these instances, the containing part can then delete the facet, because it is no longer needed.

Your part deletes the facet of an embedded frame by calling the containing facet's RemoveFacet method. (If the frame that is no longer visible has more than one facet, you need to iterate through all facets, removing each one that is not visible). OpenDoc in turn calls the embedded part's FacetRemoved method to notify it that one of its facets has been removed.

You do not have to actually delete the facet from memory the moment it is no longer needed; you can instead mark it privately as unused and wait for a call to your Purge method to actually remove it.

Creating Frame Groups

A *frame group* is a set of display frames used in sequence. For example, a page-layout part uses a frame group to display text that flows from one frame to another. Each frame in the frame group has a sequence number; the sequence numbers establish the order of content flow from one frame into the next.

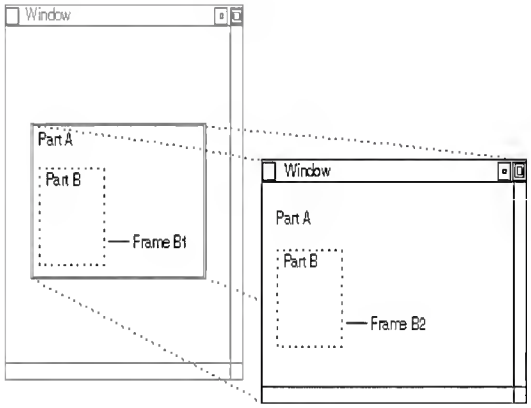
Sequence information is important for a frame group because the embedded part needs to know the order in which to fill the frames. Also your part (the containing part) needs to provide sequence information to the user, and it probably also needs to allow the user to set up or modify the sequence.

Your part creates and maintains the frame groups used by all its embedded parts. To create a frame group, you call the SetFrameGroup method of each frame that is to be in the group, passing it a *group ID* that is meaningful to you. You also assign each frame a unique *sequence number* within its group, by calling its SetSequenceNumber method. You should assign sequence numbers that increase uniformly from 1, so that the embedded part can recognize the position of each frame in the group. You can, of course, add and remove frames from the group and alter their sequence with additional calls to SetFrameGroup and SetSequenceNumber. The embedded part displayed in the frame group can find out the group ID or sequence number of any of its frames by calling the frame's GetFrameGroup and GetSequenceNumber methods.

Synchronizing Embedded Frames

If your part wants to create multiple similar views of an embedded part, you must ask the embedded part to synchronize those views. Then, if the content of one of the frames is edited, the embedded part will know to invalidate and redraw the equivalent areas in the other frames.

Frame synchronization is necessary because each display frame of a containing part represents a separate display hierarchy. For invalidating and redrawing, OpenDoc itself maintains no direct connection between embedded frames in those separate hierarchies, even if they are exact duplicates that show the same embedded-part content. the figure in Synchronizing Embedded Frames shows an example. Part A (in frame view type) is opened into a part window. Embedded frame B2, as displayed in the part window, is a duplicate of embedded frame B1 in the original frame. However, unless you synchronize the frames, Part B will not know to update the display of B1 if the user edits the content of B2.



Your part (the containing part) makes the request to synchronize frames by calling the embedded part's AttachSourceFrame method. You should call AttachSourceFrame as soon as you create the frame that needs to be synchronized with the source frame-that is, before you add any facets to the frame. How an embedded part responds to AttachSourceFrame is described in Grouping Display Frames. See Synchronizing Display Frames for a description of how an embedded part responds to AttachSourceFrame.

Transmitting your Container Properties to Embedded Parts

When one part is embedded in another part of the same or similar part category, the user may prefer that, by default, they share certain visual or behavioral characteristics. For example, if the user embeds a text part into another text part, it might be more convenient for the embedded part to adopt the text appearance (font, size, stylistic variation) of the containing part.

OpenDoc supports the communication necessary for this process by defining the concept of *container properti*es. The containing part defines whatever characteristics it wishes to transmit to its embedded parts; embedded parts that understand those container properties can choose to adopt them. Container properties are passed from the containing part to its embedded part in a storage unit; the embedded part reads whatever container properties it understands from the storage unit, and adopts them for its own display if appropriate.

Note: Container properties are defined exclusively by individual parts and may apply to only a portion of a part's content. They are therefore generally unrelated to the *properti*es of the part's storage unit, as described in Storage-Unit Organization.

As the containing part, your part can define whatever container properties it wishes to provide for adoption by embedded parts. Only parts that understand the formats of your container properties, of course, can adopt them.

An embedded part learns what container properties it might adopt from your part by calling your part's AcquireContainingPartProperties method. This is its interface:

```
ODStorageUnit AcquireContainingPartProperties(in ODFrame frame);
```

You should return your container properties to the caller in a storage unit.

Whenever your part changes any of the container properties that it expects embedded parts to adopt, it should notify each embedded part (other than bundled parts) of the change by calling the embedded part's ContainingPartPropertiesUpdated method.

Drawing

This is the second of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes how your part draws itself.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. You should also be familiar with the discussion of frames and facets presented in Frames and Facets. For additional concepts related to your part editor's runtime environment, see OpenDoc Run-Time Features. The discussion in this chapter also assumes that you are familiar with the platform-specific graphics system and drawing commands you need to draw your parts' content. For OS/2, this means the OS/2 Graphic Programming Interface (GPI).

This chapter shows how your part:

- Uses canvases as drawing destinations

- Use transforms and shapes to lay out its contents
- Draws itself to the screen
- Prints itself

For a discussion of windows, the display structures within which all your drawing typically takes place, see Windows and Menus.

Canvases

Canvases are inherently platform-specific. OpenDoc canvas objects are basically wrappers for structures that differ across platforms. This section discusses how to use canvas objects regardless of the platform for which you are developing, with some OS/2-specific information provided where appropriate.

Using Canvases

The class ODCanvas is OpenDoc's wrapper for a platform-specific (or graphics-system-specific) object that represents a drawing environment. A drawing canvas can refer to anything from a bit map or a structured display list to a stream of PostScript code. It represents the destination for drawing commands, the environment for constructing a rendered image; it has a coordinate system and may retain state information (such as pen color) that influences how drawing commands are interpreted.

A canvas object holds a reference to a system-specific object called a platform canvas, and that object is *not* deleted when the canvas is released. If you create a canvas, you must create the platform canvas separately, and you are responsible for deleting it when the canvas is deleted.

On some platforms, the platform canvas is a system-defined type or structure. On OS/2, it is a SOM class, which you create using the ODFacet method CreatePlatformCanvas.

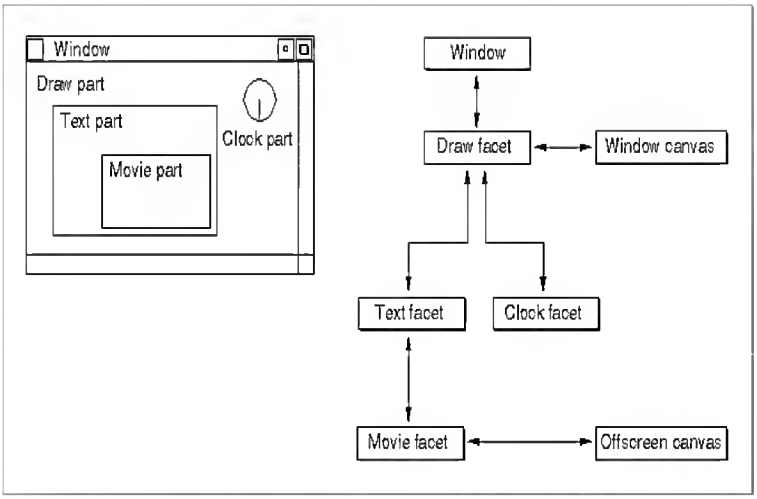
Canvas Features

Canvases can be dynamic or static, meaning that they are used for video display or printing display, respectively. Your part editor can determine whether it is drawing to a static or dynamic canvas and can adjust its procedures accordingly.

Canvases can be onscreen or offscreen. Your part editor can create special effects or improve performance by drawing a complex image to an offscreen cache and then quickly transferring the completed image to the screen. For added convenience, offscreen canvases maintain clipping and updating information that mirrors their onscreen equivalents.

Canvases are attached to individual facets. In the following figure, for example, which shows the same document with the same facet hierarchy as in the figure in Frame and Facet Hierarchies, the document has two attached canvases:

- An offscreen canvas is attached to the movie part's facet.
- An onscreen canvas is attached to the root part's facet. (A canvas attached to the root facet of a window is also called a window canvas; every window has a window canvas, which is assigned to its root facet by OpenDoc when you first register the window).



If a particular facet in a window's facet hierarchy has an attached canvas, all of its embedded facets (and their embedded facets, and so on) draw to that canvas. Thus, for most drawing, only a window's root facet needs a canvas. In the previous figure, for example, any drawing done to the text facet, draw facet, or clock facet is rendered on the window canvas.

However, if a particular part needs an offscreen canvas, for itself or for an embedded part, it can attach a canvas to a facet anywhere within the hierarchy. Any drawing done to the movie facet in the previous figure, for example, is rendered on the offscreen canvas.

On OS/2, a window canvas consists of a Presentation Manager (PM) window hierarchy that parallels the facet hierarchy. There is one PM window for each facet in the window canvas.

Every canvas has an *owner* , a part that is responsible for transferring images drawn on its canvas to the parent of that canvas. In the previous figure, for example, the movie images drawn to the offscreen canvas must be transferred to the window canvas in order to be viewed onscreen. The owner decides when and how to transfer images from a canvas to its parent. The owner of the offscreen canvas in the previous figure might be the movie part or the text part that contains the movie part.

Adding and Removing Canvases

This section describes how to create and delete canvases for your parts. For specific information on using canvases for offscreen drawing, see Offscreen Drawing. If you want to create a canvas and attach it to a facet, take these steps:

1. Create and initialize a GPI presentation space and associate it with a memory DC.
2. Create an ODPlatformCanvas object using the CreatePlatformCanvas method of ODFacet.
3. Create a canvas object, using the CreateCanvas method of ODFacet or ODWindowState. In calling the method, you assign the platform canvas to the new canvas, and you also define the canvas as static or dynamic and onscreen or offscreen. (These values cannot change for the lifetime of the canvas).
4. Designate your part as owner of the canvas by calling the SetOwner method of the canvas.
5. Assign the canvas to a facet. Because only the owner of a canvas can remove it from a facet, the timing of assigning the canvas is important:
    - If your part is a containing part assigning an offscreen canvas to one of its embedded parts, you should assign the canvas when you first create the embedded facet that is, when you first call the CreateEmbeddedFacet method of the embedded part's containing facet. Otherwise, the embedded part may assign a canvas to the facet, precluding you from doing so.
    - If your part is a containing part and one of its embedded parts has an existing facet with no assigned canvas, you can add one by calling the ChangeCanvas method of the embedded facet. When you do so, OpenDoc communicates the change to all embedded parts that use that facet, by calling their CanvasChanged methods.
    - Your part can add a canvas to any of its own display frames' facets at any time, as long as the facet does not already have an assigned canvas. It is probably best to attach the canvas as soon as the facet is created, by calling ChangeCanvas from within your part's FacetAdded method. If the containing part of your part has already attached a canvas to your new facet, you cannot assign a different canvas to it.
    - If your part absolutely needs to attach its own canvas to a facet that already has an assigned canvas, you can get around this restriction by creating a subframe of the facet's frame, creating a facet for that frame, and assigning the canvas to that facet.

To remove a canvas from a facet, take these steps:

1. Call the facet's ChangeCanvas method, passing it a null value for the canvas reference.
2. Delete the PS and memory DC that the platform canvas had referenced.
3. Delete the canvas object, by calling delete (in C++).
4. Delete the platform canvas object, by calling delete in C++.

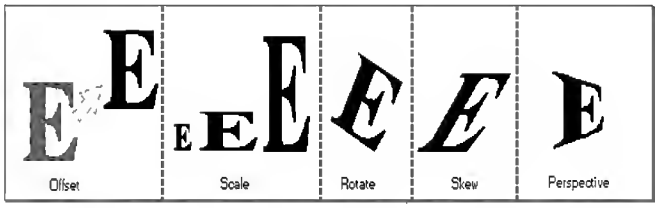
Transforms and Shapes

Unlike windows and canvases, transforms and shapes are not platform specific. The OpenDoc objects that represent them may encapsulate platform-specific structures, but they also often give added capabilities. This section discusses how you use transforms and shapes to position and clip embedded parts for drawing.

Transforms and Coordinate Spaces

Both frames and facets employ transforms. It is useful to discuss them in terms of the coordinate spaces they define.

In OpenDoc, an object of the class ODTransform is a 3x3 matrix that is used to modify the locations of points in a systematic manner. OpenDoc transforms support the full range of two-dimensional transformations shown in the following figure.

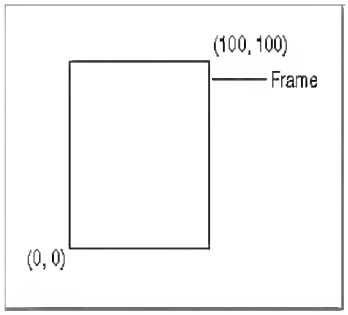


Not all graphics systems support all the features of OpenDoc transforms. Some graphics systems support only offset, or translation, of points; others support offset plus scaling; still others support all transformations. Depending on the graphics system your part editor uses, it may have to do extra work on its own to support features such as scaling or rotation. In such a case, you can create a subclass of ODTransform, if desired, that adds those features or even other, more sophisticated transformational capabilities.

A transform can be thought of as defining a coordinate space for the items that it is applied to. OpenDoc uses transforms to convert among two kinds of coordinate space: frame coordinate space and content coordinate space.

Frame Coordinate Space

The *frame coordinate space* is the coordinate system defined by the specification of the frame shape. The frame shape is the basis for the layout and drawing of embedded parts. Suppose, for example, that a frame shape is defined as a rectangle with coordinates of (0, 0) and (100, 100). In a coordinate system with the origin at the lower left as in OS/2, the shape would appear as shown in the figure below.



All shapes and geometric information passed back and forth between embedded parts and their containing parts are expressed in terms of the frame coordinate space.

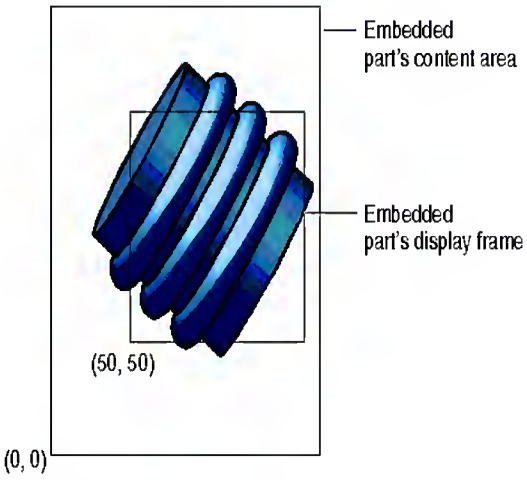
Note that the shapes describing the facet associated with a frame are described in the same coordinate space as the frame; that is, they are in frame coordinates. Thus, in this example, if the facet corresponded exactly to the frame (which is common), it would have coordinates of (0, 0) and (100, 100).

Content Coordinate Space

The co*ntent* coo*rdinate space* of a part is the coordinate system that defines locations in the part's content area. It is the local coordinate space of the part itself and typically has its origin in the lower-left corner of the part's content.

The *internal transform* of a part's display frame defines the scrolled position (as well as the scaling, rotational, and skew properties) of the part's contents within the frame. Applying the display frame's internal transform to a point in content coordinate space converts it to frame coordinate space. Conversely, applying the inverse of the internal transform to a point in frame coordinate space converts it to content coordinate space.

For example, suppose that the following figure shows the entire contents of a part, and that a portion of it is to be displayed in a frame. If the part's display frame has the dimensions shown previously (the figure in Frame Coordinate Space), and if the frame's internal transform specifies an offset value of (50, 50), the frame will appear in relation to its part's content as shown in the following figure. Only the portion of the part within the area of frame shape will be displayed when the part is drawn.



Application of the internal transform, in this case, means that a point at (50, 50) in content coordinates-the lower-left corner of the display frame-is at (0, 0) in frame coordinates. Conversely, a point at (50, 50) in frame coordinates is at (0, 0) in content coordinates.

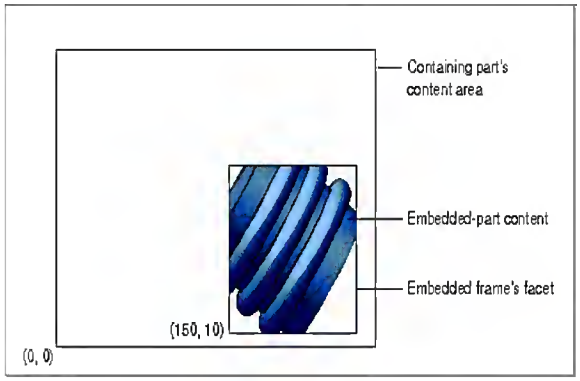
Transformations other than offsets are applied in the same manner; you can scale, rotate, or otherwise transform the contents of the part within its frame by applying the frame's internal transform.

Converting to the Coordinates of a Containing Part

Just as the internal transform of a frame defines the scrolled position of the content it displays, the external transform of that frame's facet

defines the position of the frame within its containing part.

Applying the external transform of a frame's facet to a point in frame coordinate space converts it to the content coordinate space of the containing part. For example, suppose that the facet and frame of the embedded part in the previous figure have the same shape, and suppose further that the facet's external transform specifies an offset of (150, 10). In relation to the containing part's content area, the embedded part would appear as shown in the following figure.



In this case, applying the external transform causes a point at (0, 0) in embedded-frame coordinates-the lower-left corner of the embedded part's frame-to be at (150, 10) in containing-part content coordinates.

(Conversely, applying the inverse of the embedded facet's external transform to a point in content coordinate space converts it to the embedded frame's coordinate space. Thus, in the previous figure, a point at (150, 10) in embedded-frame coordinates is at (0,0) in the content coordinates of the containing part).

Transformations other than offsets are applied in the same manner; the embedded part and its frame can be scaled, rotated, or otherwise transformed within the containing part by applying the facet's external transform.

To convert from the content coordinates of an embedded part to the content coordinates of its containing part therefore, you need to apply two transforms: the internal transform of the embedded part's display frame, followed by the external transform of that frame's facet. For the example shown in this section, you can see by inspection that the point (50, 50) in the content coordinates of the embedded part (the origin of its display frame) becomes the point (150, 10) in the content coordinates of the containing part. You could also calculate that the point (0, 0) in the contents of the embedded part (its lower-left corner) converts, by application of both transforms, to the point (100, 10) in the contents of the containing part (outside of the embedded frame and, therefore, not drawn).

You are not usually concerned with your embedded part's location within the content area of its containing part. You are, however, always interested in your content's location-and your frame's location-on the canvas or in the window in which you are drawn, as described next.

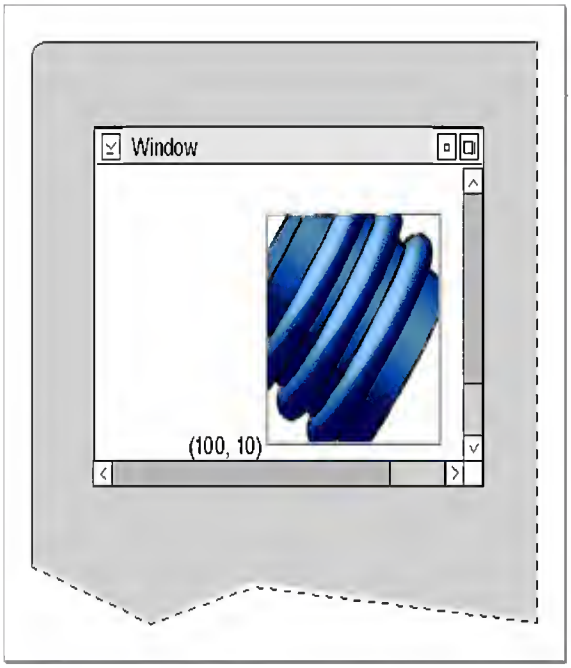
Canvas Coordinates and Window Coordinates

When your part draws its contents, or when it draws adornments to its frame such as scroll bars, it is your part's responsibility to properly position what it draws on its canvas. In setting up the canvas' platform-specific drawing environment, you need to provide information that tells the canvas where, in terms of its own coordinate space, your drawing will take place. OpenDoc does not do that for you automatically. It does, however, provide methods that make it fairly simple.

If you were to start from your part's content coordinate space and traverse the embedding hierarchy upward, applying, in turn, your part's internal transform and external transform, and then applying each internal transform and external transform of each containing frame and facet, up to a facet with an attached canvas, you arrive at the *canvas coordinate space* or the *window coordinate space* in which your part is drawn.

- If there are no offscreen canvases in the facet hierarchy, the canvas coordinate space is the same as the window coordinate space; to calculate it, OpenDoc applies all transforms up through the internal transform of the root facet. This coordinate space corresponds to the device coordinates of the window canvas.
- If there are one or more offscreen canvases in the facet hierarchy, the canvas coordinate space corresponds to the frame coordinates of the first part above yours in the hierarchy whose facet has an attached canvas. To calculate it, OpenDoc applies all transforms up through the internal transform of the frame whose facet has the canvas. The window coordinate space, in this case, is unaffected by the presence of an offscreen canvas; it is still defined by the device coordinates of the window canvas.

The following figure shows a simple example of converting to canvas coordinates and window coordinates. The containing part and the embedded part are the same ones as shown in the figure in Converting to the Coordinates of a Containing Part, and the containing part is, in this case, the root part of the window. The internal transform of the root frame (the containing part's display frame) specifies an offset of (50, 50), and the external transform of the root facet is identity.



Converting content coordinates to window coordinates means concatenating all the transforms up through the root frame's external transform. Assuming there is no offscreen canvas in the previous figure, the point (50,50) in content coordinates (the origin of the embedded part's frame, as shown in the figure in Content Coordinate Space) becomes the point (100, 10) in window or canvas coordinates. If there were an offscreen canvas attached to the embedded facet in the previous figure, the canvas coordinates for the origin of the embedded part's frame would be (0,0).

Normally, you do all your drawing in canvas coordinate space. That way, whether or not you are drawing to an offscreen canvas (which you might not control or even be aware of), your positioning will be correct. However, when you need to draw directly to the window to provide specific user feedback, as described in Drawing Directly to the Window, you need to work in window coordinates.

By concatenating the appropriate internal and external transforms, OpenDoc calculates four different composite transforms that you can use for positioning before drawing:

- The composite transform from your content coordinates to canvas coordinates is called your *content transform* , and you apply it when drawing your part's contents on any canvas. You obtain it by calling your facet's AcquireContentTransform method.
- The composite transform from your frame coordinates to canvas coordinates is called your *frame transform* , and you apply it when drawing any frame adornment on the canvas. You obtain it by calling your facet's AcquireFrameTransform method.
- The composite transform from your content coordinates to window coordinates is called your *window-content transform* , and you apply it when drawing your part's contents directly to the window. You obtain it by calling your facet's AcquireWindowContentTransform method.
- The composite transform from your frame coordinates to window coordinates is called your *window-frame transform* , and you apply it when drawing any frame adornment directly to the window. You obtain it by calling your facet's AcquireWindowFrameTransform method.

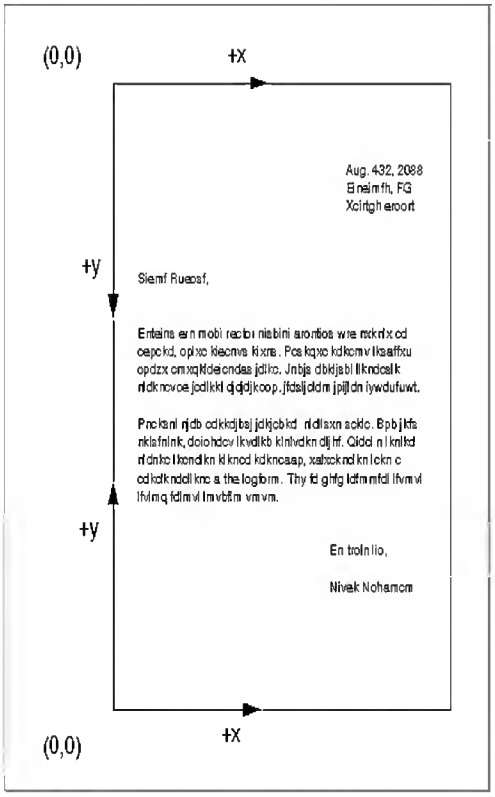
For a description of how to use these transforms when setting up for drawing, see Draw Method of your Part Editor.

Transforms and Hit-Testing

Hit-testing is, in a sense, the inverse of drawing; it involves a conversion from window coordinates to your part's content coordinates. In most circumstances, OpenDoc takes care of this for you. See Hit-Testing for more information.

Coordinate Bias and Platform-Normal Coordinates

On each platform, OpenDoc uses the platform's native coordinate system, called *platform-normal coordinate*s , for stored information and for internal calculations. For example, on OS/2, coordinates are measured with the origin at the lower-left (of the screen, of a shape, or of a page, and so on), with increasing values to the right and upward. Some other platforms use coordinates with an origin at the upper-left, with values increasing to the right and downward. The following figure shows how these two coordinate systems would apply to measurements on a text part's page.



OpenDoc functions consistently on any platform, regardless of the platform's coordinate system, without need for coordinate conversion. However, some platforms allow for simultaneous existence of different coordinate systems. In such a case, a part editor that assumes a particular coordinate system will not function correctly with OpenDoc on a platform with a different coordinate system, unless it first accounts for the *coordinate bias* , or difference between its coordinate system and platform-normal coordinates.

Bias Transforms

Coordinate bias usually takes the form of an offset in the origin, a change in the polarity of one or more axes, and possibly a change in scale. A transformation matrix, called a *bias transform* , is applied to measurements in a part's coordinate system to change them into platform-normal values.

You do not have to calculate bias transforms yourself. When you create a canvas and define its graphics system, OpenDoc uses that information to calculate a bias transform if it is required. In constructing the bias transform, OpenDoc also needs to know your part's content extent (described next).

Content Extent

To convert locations in your part's content between coordinate systems whose origins are offset, the vertical extent of your content (in essence, the height of your part's page) represents the offset between the two coordinate origins. See the figure in Coordinate Bias and Platform-Normal Coordinates for an example. This *content extent* is the offset needed in the bias transform that performs the conversion.

Because your part may be drawn at any time on a canvas that has a bias transform attached, you should always make the value of your content extent available. When a frame is added to your part, and whenever the content extent of your part changes (such as when you add a new page), you should call your display frame's ChangeContentExtent method so that the frame always stores the proper value. A caller constructing a bias transform can obtain the current content extent of your part by calling the GetContentExtent method of your part's frame.

The biasCanvas Parameter

The classes ODFrame and ODFacet include several methods, such as ChangeInternalTransform and ContainsPoint, that specify shapes or calculate positions on a canvas. Because these calculations necessarily assume a coordinate system, the methods include a parameter, biasCanvas, that allows you to specify a canvas whose attached bias transform is to be used to convert between your part editor's coordinates and platform-normal coordinates. Thus, after you set up your offscreen canvas for drawing in your own coordinate system, you can also use it to make sure that all point, frame, and facet geometry is properly converted for you.

Using Transforms

The sections that follow cover some of the basic ways you can use transforms to position and modify the content of your part and embedded parts.

Scrolling your Part in a Frame

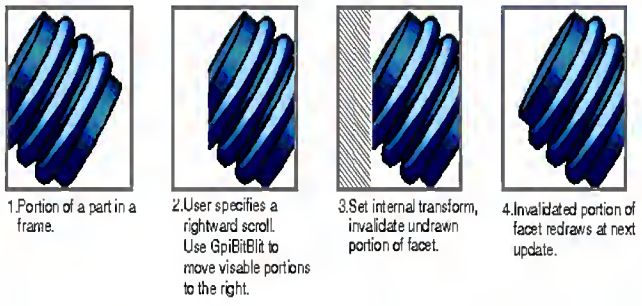
If your part's content area is greater than its frame area, only a portion of the content can be displayed in the frame. The user needs to be able to choose which portion of a part shows through its frame by moving the part's contents-as a unit-in relation to the frame position.

The standard OpenDoc method for supporting this ability involves the **Show Frame Outline** menu command, available in the View menu when the user opens your part's frame into its own part window. In this mode, the user can drag an outline of the frame, positioning it as desired in your part's content area. Another standard method involves a mode in which the user employs a hand-shaped cursor to drag the part within its frame. It is also possible to use page-up and page-down keys or other keyboard input to cause scrolling.

Providing scroll bars is another way to support scrolling. If your part is the root part of a document window or part window, scroll bars are the standard method. If your part is an embedded part displayed in a frame, scroll bars may or may not be appropriate, depending on the size of your frame and the nature of its contents.

Programmatically, changing the portion of a part displayed in a frame involves modifying the offset (translational setting) of the internal transform of the part's frame and then redrawing. If you do not have scroll bars within your frame, these are the basic steps:

1. Obtain a copy of your frame's internal transform, by calling the frame's AcquireInternalTransform method.
2. Call the MoveBy method of the transform, passing it the negative of the amount by which you want the frame to scroll.
3. Reassign the changed transform to your frame, by calling your frame's ChangeInternalTransform method.
4. Redraw the scrolled frame, as shown in the following figure and discussed in Drawing when a Part Is Scrolled.



For more information on redrawing frames that include scroll bars or other adornments, see Drawing with Scroll Bars. For more information on handling scroll bars, interpreting events in them, and redrawing frames that include scroll bars or other adornments, see Scrolling.

Transforming the Image of your Part

If the graphics system used by your part editor supports all transformations, you can simply modify the internal transform of your display frame to achieve the kinds of special effects shown in the figure in Transforms and Coordinate Spaces. By modifying the internal transform appropriately, you not only can position your part's content in its display frames, but you can also scale, rotate, skew, and apply perspective to it. The OS/2 GPI supports all types of transforms except perspective (see the figure in Transforms and Coordinate Spaces).

For example, you could change the scale of your part's content-to support either higher precision in positioning or larger total content area than would otherwise be possible-by including a scale factor in your display frames' internal transforms. Then, to leave embedded parts' displays unaffected by the scaling, you could apply the inverse scaling factor to the external transforms of all embedded facets.

Even if your graphics system does not support features such as rotation or skew, you can still "pre-transform" your images by first applying the internal transform to all drawing coordinates, before passing them to the drawing commands.

Positioning an Embedded Frame

In general, your containing part can store information on the shapes and positions of embedded frames in any format that is convenient for you. When frames become visible, however, OpenDoc needs that positioning information to allow it to dispatch events correctly and to correctly place the drawn images of each embedded part.

Thus, when you create a facet for a visible frame embedded in your part, you assign it (when calling the CreateEmbeddedFacet method) an external transform whose offset reflects the position of the embedded frame in the coordinate system of your part content. If you later reposition that frame, you need to modify its facet's external transform, if the frame is visible. You can change a facet's external transform, as well as its clip shape, by calling its ChangeGeometry method.

Transforming the Image of an Embedded Part

You can use the external transform of a facet embedded in your part to achieve special display effects, regardless of the display intention of the embedded part. By modifying the external transform appropriately, you not only can position the frame within your part's displayed content but can also scale, rotate, skew, and apply perspective to it.

For example, you could make separate embedded frames appear to be the faces of a cube, giving each the proper skew or perspective necessary to achieve the effect, regardless of what is being drawn in each frame by its own part editor.

Using Drawing-Related Shapes

A *shape* object is an OpenDoc object that is an instance of the class ODShape. It is the specification of a two-dimensional shape in a given coordinate space. Different platforms and different graphics systems define shapes differently. Objects of the class ODShape are partly wrappers for system-specific shape structures, but they also have the ability to convert among several common shape structures.

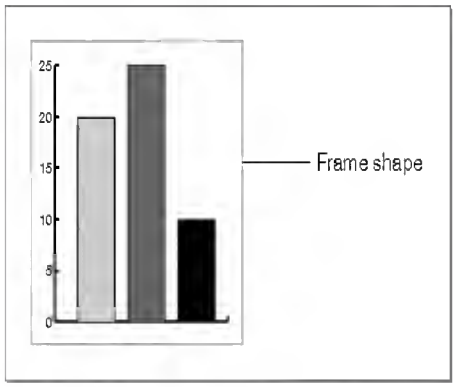
OpenDoc uses shape objects for clipping purposes in drawing and hit-testing. There are four drawing-related shapes: the frame shape and used shape, attached to the frame object; and the clip shape and active shape, attached to the facet object.

Frame Shape

The frame shape is discussed in many places in this book, including earlier in this chapter, in Frame and Facet Hierarchies. The frame shape represents the fundamental contract between embedded part and containing part for display space. The other drawing-related shapes are variations on the frame shape, and all are defined in the same coordinate system as the frame shape.

A frame shape is commonly rectangular, as shown in the following figure, but it can have any kind of outline, even irregular. An embedded part can request a nonrectangular frame shape if it has a special need for one; for example, a part that displays a clock face might request a

round frame shape. It is the containing part's right to comply with or deny that request, and it is also the containing part's responsibility to draw that frame's selected appearance, including resize handles, if appropriate.



The frame shape is defined by the containing part and is stored in the frame object. Your part (the containing part) can set an embedded frame's frame shape by calling the frame's ChangeFrameShape method. Any caller may access the frame shape by calling the frame's AcquireFrameShape method.

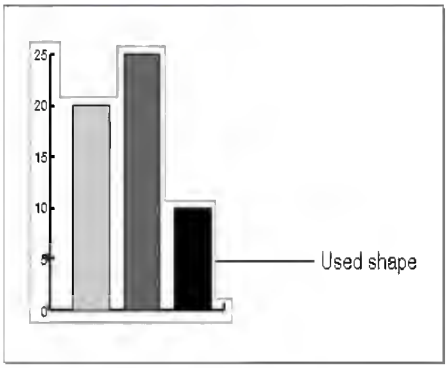
By convention, your part should use the frame shape when drawing the selected frame border of an embedded part.

Cross-platform representation

The frame shape is stored persistently, and your part may subsequently be displayed under a different graphics system. Therefore, it is important that a frame shape have a platform-neutral (that is, polygonal) representation when it is stored.

Used Shape

An embedded part may need a nonrectangular shape to draw in, may need to change frame shape often, or may want to allow the containing part to be able to draw within portions of its frame. In these cases, the embedded part need not negotiate for an unusual frame shape or continually renegotiate its frame size. Instead it can define a *used shape* to tell its containing part which portions of its frame shape it is currently using. In the following figure, for example, the embedded part retains a rectangular frame shape but defines a used shape that covers only the content elements it draws. The containing part can then use that information to, perhaps, wrap its content to the used portions of the embedded part. For example, the third figure in Event Handling shows a containing part (a text part) that wraps its text closely to the used shape of an embedded part (a pie-chart part whose frame shape is actually rectangular).



The used shape is defined by the embedded part, specified in frame coordinates, and stored in the frame object. Any caller may access the used shape by calling the frame's AcquireUsedShape method. If the embedded part has not stored a used shape in its frame, AcquireUsedShape returns the frame shape as a default.

Note that it does not make sense for the used shape to extend beyond the edges of the frame shape, because the clip shape (described on Clip Shape) is based on the frame shape and no drawing occurs outside of the clip shape.

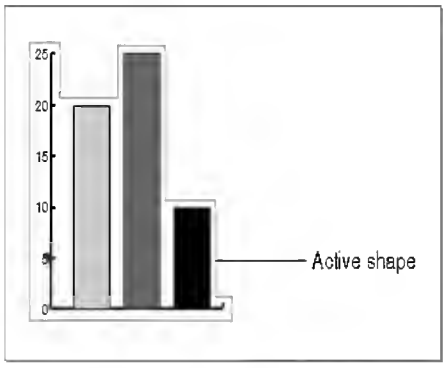
Your part can set its display frame's used shape by calling the frame's ChangeUsedShape method. (You can pass a null shape to ChangeUsedShape to make your used shape identical to your frame shape. If you do so, be sure to call ChangeUsedShape again whenever your frame shape changes.) When you call ChangeUsedShape, OpenDoc then calls the UsedShapeChanged method of your containing part:

```
void UsedShapeChanged(in ODFrame embeddedFrame);
```

If your part is the containing part in this situation and has wrapped its content to the embedded part's used shape, you can use this method to adjust your content to the new used shape.

Active Shape

A facet's *active shape* is the area within a frame in which the embedded part is willing to receive geometry-based (mouse) events. The active shape of a facet is often identical to either the frame shape or the used shape of its frame, as shown in the following figure. The active area of a part is likely to coincide with the area it draws in; events within the frame shape but outside of the used shape are better sent to the containing part, which may have drawn in that area.



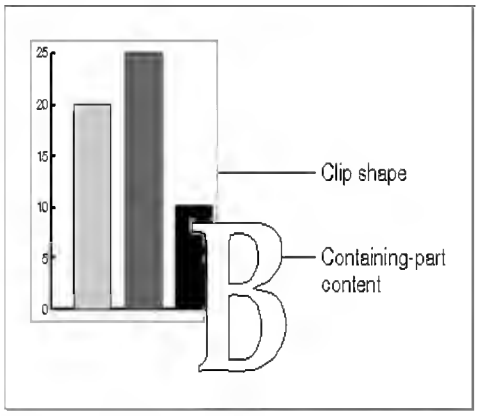
The active shape is defined by the embedded part, is specified in frame coordinates, and is stored in the facet object. Your embedded part can set the active shape of its display frame's facet by calling the facet's ChangeActiveShape method. Any caller may access the active shape by calling the facet's AcquireActiveShape method. If the embedded part has not stored an active shape in its facet, AcquireActiveShape returns a copy of the frame's shape.

Note that the effective active shape-the active shape as the user perceives it-is the intersection of the active shape and the clip shape (described next). Events within the area of obscuring content or other frames that block the active shape are not passed to the embedded part.

OpenDoc uses the active shape when drawing the active frame border of an embedded part.

Clip Shape

A facet's *clip shape* defines the portion of a frame that is actually to be drawn. The clip shape of a facet is commonly identical to the frame shape of its frame, except that it may be modified to account for obscuring content elements or overlapping frames in the containing part, as shown in the following figure.



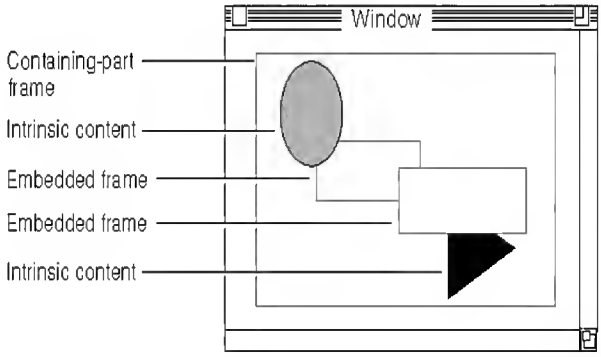
The clip shape is defined by the containing part, it is specified in frame coordinates, and it is stored in the facet object. Your part (the containing part) can set the clip shape of an embedded frame's facet by calling the facet's ChangeGeometry method. Any caller may access

the clip shape by calling the facet's AcquireClipShape method.

Managing Facet Clip Shapes

This section describes how your part adjusts the clip shapes of its embedded frames' facets before drawing, to account for overlapping relationships among embedded facets and elements of intrinsic content.

Sibling frames are frames embedded at the same level within a containing frame. They may be frames in a frame group (described in Creating Frame Groups), or they may be unrelated frames belonging to different embedded parts. Because sibling frames can overlap each other and can overlap (or be overlapped by) intrinsic content of the containing part, it is the responsibility of the containing part to ensure that clipping occurs properly. The following figure shows examples of the overlapping relationships that can occur.



Maintaining a List of Embedded Facets

If your part contains embedded frames, it must maintain a z-ordered (front-to-back) list of the embedded frames and content elements, so that you can reconstruct the overlapping relationships among them. You use that list to update the clip shapes of the facets of your embedded frames.

You control the z-ordering among sibling frames embedded in your part, and you can communicate that information to OpenDoc through the MoveBehind and MoveBefore methods of the containing facet of the sibling frames' facets. The facet positioning you achieve with MoveBehind and MoveBefore is reflected in the order in which you or OpenDoc (or any caller) encounters facets when you use a *facet iterator* (class ODFacetIterator) to access each of the sibling facets in turn (as, for example, when calculating a resulting clip shape).

Calculating the Clip Shapes

If your part contains embedded frames, it performs two related tasks when managing its embedded facets' clip shapes.

- It must clip all of its embedded facets so that sibling embedded parts don't improperly overwrite each other, the containing part's intrinsic content, or the active frame border. (Managing the Active Frame Border describes how to account for the active frame border).
- It must clip its own intrinsic content so that it doesn't improperly overwrite embedded parts or the active frame border. This also means taking responsibility for drawing in the area of an embedded frame that is outside the embedded frame's used shape. Because the embedded part does not draw outside of its used shape, it is the containing part's responsibility to do so.

In calculating the clip shapes for your embedded parts' facets, remember these points:

- A facet's clip shape is affected only by sibling elements, whether they be items of intrinsic content or facets of other embedded frames. Your own display facet's clip shape, and the clip shapes of facets embedded within your embedded parts, have no effect.
- When one embedded facet obscures another, use the used shape of the obscuring facet's frame to clip the obscured facet.
- When intrinsic content obscures an embedded facet, make sure to account for all of it-including selection handles, the active frame

border, and other adornments.

- You need to recalculate your embedded parts' clip shapes whenever a content element or an embedded frame is added, deleted, moved, adorned with resize handles, or modified in any way that affects how it is drawn. However, when a given item changes, you need only to recalculate the clips of it and the elements *behind* it; the clipping of elements lower in z-order is unaffected.

Your routine to recalculate embedded-facet clip shapes could take steps similar to these:

1. Create a "working clip," a clip shape that at each point in the calculations represents the total area that is not obscured of your display facet. Start it as a copy of your own display frame's used shape; only content elements within that area need to have their clip shapes recalculated.

2. If the currently active frame is embedded in your part, subtract the active frame border from your working clip. (See Managing the Active Frame Border).

3. Iterate through all your z-ordered content elements, front to back. For each element that is an item of intrinsic content, do this:

    - Calculate a new clip shape, representing the visible portion of the item, by intersecting the item's shape with the current working clip. Store the new clip shape according to your own content model.

    - Calculate a new mask shape that includes both the (clipped) content item shape and any adornments (such as selection handles) it may have. Modify the working clip by subtracting that mask shape from it, so that elements behind this content item will be obscured.

    Likewise, for each element that is an embedded facet, do this:

    - Get the frame shape for the facet's frame and convert it to your own content coordinates. Intersect it with the current working clip (to account for obscuring content in front), convert it back to embedded-frame coordinates, and assign it as the facet's new clip shape.

    - Calculate a new mask shape by getting the used shape for the facet's frame and converting it to your own content coordinates. Modify the working clip by subtracting that mask shape from it, so that elements behind this embedded frame will be obscured.

Embedded-Part Responsibilities

The containing part is responsible for its embedded facets' clip shapes, but the embedded part has some responsibilities, to ensure that drawing occurs properly:

- The embedded part must always clip its own drawing operations correctly, using its aggregate clip shape; see Draw Method of your Part Editor.

- The embedded part must never draw outside of its used shape. Only a root part can draw outside of its used shape.

- When a containing part changes the clip shape of one of its embedded frame's facets, OpenDoc notifies the embedded part of the change by calling the embedded part's GeometryChanged method. If the embedded part is drawing asynchronously (see Asynchronous Drawing), its next asynchronous draw must use the new clip shape.

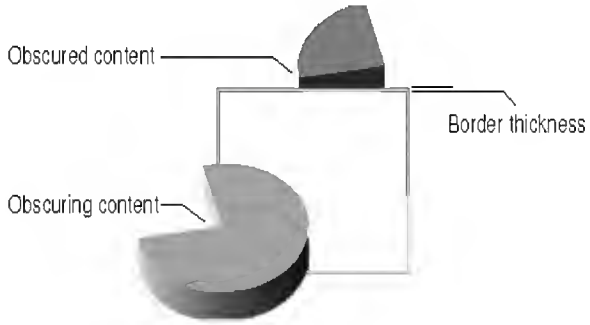
Managing the Active Frame Border

When a partcsq.s frame is active, OpenDoc draws the active frame border (shown in the following figure) around the frame. Neither the active part itself nor its containing part need to draw the border.

However, your active part may need to adjust the clipping of the border shape and also adjust the clipping of its own content to account for the border. Furthermore, your part may need to suppress the drawing of the active border in special instances, such as when creating split-frame views. This section discusses both tasks.

Adjusting the Active Frame Border

The active frame border occupies an area a few pixels wide in the content area of the active partcsq.s containing part, and it may overlap or be overlapped by other content elements (embedded frames or intrinsic content) of the containing part (see the following figure). Therefore, it is up to the containing part to make sure that the active frame border is appropriately clipped by elements in front of the active frame, and that elements behind the active frame are appropriately clipped so that they do not draw themselves within the border area.



When a new frame acquires the selection focus, or when the active frame changes its frame shape, OpenDoc calculates a new active border shape. It then passes that shape to the active part's containing part, by calling the containing part's AdjustBorderShape method:

```
ODShape AdjustBorderShape(in ODFacet embeddedFacet,
                          in ODShape shape) ;
```

Your AdjustBorderShape method has two tasks: clip the active border shape to account for obscuring content, and clip any of your own content that is obscured by the border.

When the active frame embedded in your part becomes inactive and thus no longer has the active border around it, OpenDoc notifies your part of the change by calling your part's AdjustBorderShape method once more, this time passing a null value for the shape parameter. You can then remove the clipping you had previously applied to your obscured content.

Note: If your part receives several consecutive calls to its AdjustBorderShape method, it means that your embedded (active) frame has several facets. Consider the active border shape to be the union of all the provided shapes.

In your AdjustBorderShape method, you can take steps such as the following. These steps assume that you maintain a field in your part that is a copy of the clipped active frame border, and that your embedded-frame clipping method (see Managing Facet Clip Shapes) uses that active-border field to clip your content elements.

1. If the shape parameter is null, you no longer need to maintain an active frame border. Release the border shape you had maintained, set your active-border field to null, call your embedded-frame clipping method to remove the clipping that had been caused by the presence of the active frame border, and exit.

2. If the shape parameter is non-null, copy the shape, convert it to your own content coordinates, and intersect it with your own used shape, so that the active border will not be drawn outside of your used shape.

3. To calculate the resultant clipped border shape, iterate through all your z-ordered content elements, starting from the facet of the active frame and working toward the front. (Only items in front of the facet can clip it).

    - For each element that is an item of intrinsic content, calculate a mask shape that includes both the item's shape and any adornments (such as selection handles) it may have. Subtract that mask shape from your active-border shape.

    - For each element that is an embedded facet, calculate a mask shape by getting the used shape for the facet's frame, converting it to your own content coordinates, and subtracting it from your active-border shape.

4. If your active-border field is currently null, place the resulting active-border shape in it. If it is currently non-null, you are calculating a composite border shape from several facets of the active frame; in that case, replace the shape already in the field with the union of itself and the resulting active-border shape.

5. Call your embedded-frame clipping method (see Managing Facet Clip Shapes), so that it will recalculate the clip shapes of all items behind the active frame, making sure that they don't overwrite it.

If your AdjustBorderShape method does nothing and simply passes back the shape it receives, it must increment the shape's reference count before returning it.

Using Subframes to Suppress the Active Frame Border

OpenDoc draws the active frame border (see, for example, the figure *Inactive, Active, and Selected States of an Embedded Part* in Activation

and Selection) around a frame of the currently active part. (It actually follows the active shape of the frame's facet, as described in Active Shape.) When your part is the active part, OpenDoc automatically draws the active frame border around your part's display frame (the frame that has the selection focus).

Situations may occur in which you do not want the active frame border around your part's frame, even when it is active. For example, your part's frame may represent a single pane in a split window (see Using Multiple Facets). In this case, the entire window should be considere active, not just a single pane.

In such a case you can force OpenDoc to draw the active frame border around your display frame's containing frame instead of your display frame itself. You can call your display frame's SetSubframe method to assign it as a subframe of its containing frame. (A frame and a subframe must display the same part.)

You can test whether a frame is a subframe of its containing frame by calling its IsSubframe method.

Drawing

Fundamental to OpenDoc is the responsibility of each part in a compound document to draw itself, within the limits of the frame provided by its containing part. Drawing typically occurs when your part editor is asked to draw a particular facet of a particular frame in which its part is displayed. Your part editor is responsible for examining that facet and frame and displaying the correct data, with the appropriate representation, properly transformed and clipped.

Before drawing, your part editor should, if necessary, update the used shapes of its frames and the active shapes of its facets, so that the containing part can lay itself out correctly and so that only the proper events are dispatched to your part by the document shell.

This section begins by discussing how your part defines the general characteristics of its display and outlining several aspects of the basic drawing process. The section then discusses:

- Drawing with scroll bars
- Drawing directly to the window
- Asynchronous drawing
- Drawing offscreen
- Use of multiple frames and facets
- Storing cached representation of your frames' content

Defining General Display Characteristics

OpenDoc parts can display themselves in different ways in different frames or in different ways in the same frame at different times. The part is in control of its display within the borders of its frames, but there are conventions for other parts to request that the part display itself in a particular way.

There are two kinds of display categories. The *view type* of a frame indicates whether the part within it is shown as one of several kinds of icons (standard icon, small icon, or thumbnail) or whether the part content itself is drawn within a frame. Some view types are standard values defined by OpenDoc; any part should be able to draw itself in any of the standard view types. You can define your own view types also.

The *presentation* of a frame describes, for parts whose view type is framed, how the content is to be represented within the frame. Presentations are defined by parts. You define what types of presentations your part supports and you assign their values. Examples of presentations are top view, side view, wireframe, full rendering, table, bar chart, pie chart, and tool palette.

View type and presentation are represented as tokenized ISO strings of type ODTypeToken. If you create a presentation type, you define it first as an ISO string and then use the session object's Tokenize method to convert it to a token.

View Type

You can get and set the view type of your own frames by calling the frame's GetViewType and SetViewType methods, respectively.

In general, a part is expected to adopt the view type specified by its containing part. However, another part (such as your part's containing part) can change your frame's view type at any time, and you can change the view type of a frame of another part (such as one of your part's embedded parts) at any time, by calling the frame's ChangeViewType method. ChangeViewType sets the view type and then notifies the owning part, by calling the ViewTypeChanged method:

```
void ViewTypeChanged(in ODFrame frame);
```

If your part receives this call, it should display itself in the specified frame according to the indicated view type. Parts must support all standard view types. If for some reason your part receives a request for a view type that it does not support, call your display frame's SetViewType method on your display frame to change it back to a view type that you support. (Calling SetViewType does not result in a call to your ViewTypeChanged method).

Note that a change in view type can mean a change in the optimum size for your display frame. Your ViewTypeChanged method might therefore call your display frame's RequestFrameShape method at this point to initiate frame negotiations.

Presentation

You can get and set the presentation of your own frames by calling the frame's GetPresentation and SetPresentation methods, respectively.

Note that another part can change your frame's presentation, or you can change the presentation of another part's frame, by calling the frame's ChangePresentation method. In response, ChangePresentation sets the presentation and then notifies the owning part, by calling the part's PresentationChanged method:

```
void PresentationChanged(in ODFrame frame);
```

If your part receives this call, and if it supports the indicated presentation, it should display itself in the specified frame accordingly. If it recognizes the indicated presentation but does not support it, or if it does not recognize it, your part should instead pick a close match or a standard default. It should then call the frame's SetPresentation method to give it the correct value. (Calling SetPresentation does not result in a call to your PresentationChanged method).

Part Info

The part info data of a frame is a convenient place for your part to store information about that particular view of itself. The information can be anything from a simple ID to a pointer to a complicated structure or a reference to a helper object. A frame's part info is stored with the frame, not the part. Thus, if the part has many frames and if only a few have been read into memory, only the part info of those frames will be taking up space in memory.

To assign information to a frame's part info field, use its SetPartInfo method; to retrieve its part info, use its GetPartInfo method. Writing a frame's part info to its storage unit and reading it back in are described in Reading and Writing Part Info.

The facet object also contains a part info field, which can hold any kind of display-related information that you wish, such as transforms, color information, pen characteristics, or any other imaging-related data.

To assign information to a facet's part info field, use its SetPartInfo method; to retrieve its part info, use its GetPartInfo method. (A facet's part info is not persistently stored, so there are no methods for reading and writing it).

Basic Drawing

This section discusses how your part editor performs its drawing tasks in typical situations.

Setting Up

To set itself up properly for display, your part should have previously examined the following information in the given facet and its frame:

- The view type of the frame tells your part whether it should display itself as an icon or whether it should show its contents. Your

part should be ready to display itself in the specified frame according to the frame's view type.

- The presentation of the frame tells your part what kind of presentation (as defined by your own part) its content is to have, if it has frame view type. If your part does not support the frame's presentation, substitute a default presentation and call the frame's SetPresentation method to give it the correct value.

- Your part may be within a selection in its containing part. If so, it may need to be highlighted appropriately for the selection model of the containing part; see Drawing Selections for more information. You can call the GetHighlight method of your facet at any time to see whether you should draw your content with full highlighting, dim (background) highlighting, or no highlighting.

  Alternatively, when your part's HighlightChanged method is called, it can record the style of highlighting it has been assigned. It can then in turn call the ChangeHighlight method of all facets embedded in the facet whose highlight has changed, to make sure that they reflect the same highlight style.

- Check to see if you need to draw borders around any link sources and destinations in your part's content, as described in Link Borders.

- Both the frame and the facet may have information in their part info fields, placed there by the part itself, that can provide additional display-related information or objects. Use the GetPartInfo method to obtain it.

- Your part should also be aware of the kind of canvas on which it is being drawn. You can call the GetCanvas method of the facet to get a reference to the drawing canvas.

  If the canvas is dynamic (that is, if the result of calling the canvas's IsDynamic method is kODTrue), the part is being drawn onto an interactive device like a video display. Otherwise, it is being drawn to a static device like a printer (or perhaps a print-preview image onscreen). Your part may display its content differently in each case; for instance, it might display scroll bars only on dynamic canvases, and it might engage in frame negotiation if it is being drawn on a static canvas. For considerations about static canvas, see Printing.

  Your part can make these adjustments from within its FacetAdded and CanvasChanged methods. When a facet is added, the static/dynamic nature of its canvas is fixed and cannot be changed unless the canvas itself is changed.

Draw Method of your Part Editor

Your part must be able to display its content in response to a call to its Draw method:

```
void Draw(in ODFacet facet, in ODShape invalidShape);
```

When it receives this call, your part editor draws the part content in the given facet. Only the portion within the update shape (defined by the invalidShape parameter) needs to be drawn. The update shape is based on previous Invalidate calls that were made involving this facet or containing facets, or on updating needed because of part activation or window activation. The shape is expressed in the coordinate system of the facet's frame.

Your part should take steps such as these to draw itself:

1. Your part must make sure that any required platform-specific graphics structures are prepared for drawing and that the drawing context for the facet is set correctly. You must at least do the following:

   - Obtain a handle to the presentation space to be used for drawing. You do this by calling the canvas' GetPlatformCanvas method to obtain the platform canvas object for the canvas and then call the GetPS method of the platform canvas.

   - Set the default viewing transform of the presentation space. Call your facet's AcquireContentTransform method to get the proper conversion from your part's content coordinates to canvas coordinates, convert the transform to a MATRIXLF structure and pass the MATRIXLF structure to the GpiSetDefaultViewingTransform function.

   - Set the clip region for drawing your part. Call your facet's AcquireAggregateClipShape method to transform and intersect your facet's clip shape with all enclosing clip shapes, resulting in a composite clip shape in canvas coordinates. Then, transform the aggregate clip shape to canvas coordinates and call the shape object's GetRegion method to obtain a GPI region handle. Use this handle to set the clip region with the GpiSetClipRegion function.

     OpenDoc provides a utility library (FocusLib) that facilitates this step for the OS/2 platform. The source code for FocusLib is provided with the OpenDoc Programmer's Toolkit.

2. If your part has placed embedded parts on an offscreen canvas, have the embedded parts draw themselves first; see Updating an Offscreen Canvas.

3. Your part can then draw its contents, using platform-specific drawing commands.

Your part can alter the normal order of drawing, perhaps to combine overlapping embedded parts and content elements combine to achieve a specific visual effect.

- You can force the drawing of a given embedded part at any point by calling the embedded facet's Draw method.
- Calling the embedded facet's DrawChildren method in addition forces the embedded part's own embedded parts to draw themselves immediately.
- Calling the embedded facet's DrawChildrenAlways method forces the embedded part's embedded parts to draw all of their content immediately, whether or not the content has been invalidated.
- From within your parts draw method, you may also call the update method of your embedded facets, specifying KODNULL for the invalid shape. This will cause the embedded facet, and any of its embedded facets, and so on, to be updated as needed based on the current invalid shape.

Note: Unless you are the root part drawing in the root frame, never draw outside of the used shape of your display frame. Your part's containing part may be wrapping its content to the edges of your used shape.

Invalidating and Validating your Content

To mark areas of your part's content that need redrawing, you modify the update shape of the canvas on which your part is imaged. To do so, you call the Invalidate method of your display frames or their facets, passing a shape (invalidShape), expressed in your own frame coordinates, that represents the area that needs to be redrawn. OpenDoc adds that shape to the canvas's update shape, and when redrawing occurs again, that area is included in the invalidShape passed to your Draw method.

Likewise, to remove previously invalid areas from the update shape of the canvas, you call the Validate method of your frame or facet, passing a shape (validShape) that represents the area that no longer needs to be redrawn.

Validating is not a common task for a part editor to perform; OpenDoc automatically validates all areas that you draw into with your Draw method, and when you draw asynchronously, you automatically validate the areas that you have redrawn when you call the DrawIn method of the facet.

Drawing on Opening a Window

When a window opens, the facets for all the visible parts of the document are created when the window's Open method is called. The root part's FacetAdded method is called; it builds facets for its embedded frames and invalidates them. OpenDoc, in turn, calls the embedded parts' FacetAdded methods, so that they can build facets for their own embedded parts, and so on.

Each part editor whose frame is visible in the window (that is, whose frame has an associated facet) thus receives a call to its Draw method. The method draws the contents of its frame into the facet, by either using privately cached presentations or making standard drawing calls, taking into account the frame's and facet's aggregate transforms and clip shapes.

Drawing when a Window Is Scrolled

When the entire contents of a window is scrolled, the scrolled position of the contents is determined by the value of the internal transform of the root part's frame. If any of its embedded frames are made newly visible or newly invisible by scrolling, the root part should create or destroy their facets by calling the root facet's CreateEmbeddedFacet or RemoveFacet method, as described in Adding a Facet and Removing a Facet.

To force redrawing of those parts of the window that need to be redrawn after scrolling, the root part marks the area that needs to be redrawn as invalid, by calling the root facet's Invalidate method, so that it will be redrawn when an update event occurs.

Redrawing a Frame when It Changes

ed per Chuck Dumont 12/8/95 - Diane Lovelett If your part changes an embedded frame's shape, you may need to refresh its presentation as follows:

1. Change the frame's shape, as appropriate, by calling ChangeFrameShape. OpenDoc then calls the embedded part's FrameShapeChanged method, passing it the new frame shape.

   (Your part may receive a call to its UsedShapeChanged method as a result of changing the embedded frame's shape.)

2. Change the clip shape-and external transform, if necessary-of the frame's facet to correspond to the new frame shape and, possibly, new used shape.

3. If there is an undrawn area outside of the new facet shape, invalidate the portions of your own intrinsic content that correspond to the difference between the old and new frame shapes.

When the next update event is generated, the Draw methods of the embedded part and your part, as appropriate, are called to draw the invalidated areas.

Drawing when a Part Is Scrolled

When a user scrolls the contents of an embedded part, the containing part takes no role in the repositioning or redrawing. The embedded part sets its frame's internal transform to the appropriate value, as discussed in Scrolling your Part in a Frame, and calls the facet's Invalidate method to force the redraw.

If you are the part whose contents are to be scrolled, take these steps:

1. Call your frame's SetInternalTransform method, to set the transform's offset values appropriately. How you obtain the offset values you pass to SetInternalTransform depends on what kinds of events you interpret as scrolling commands and how you handle them. For example, handling events in scroll bars within your display frames is discussed in Scrolling.

2. If any embedded frames become visible or invisible as a result of the scrolling, create or delete facets for them as described in Adding a Facet and Removing a Facet.

3. Call your facet's Invalidate method to force a redraw at the next update event. You should be able to shift the bits of your frame's image by the scrolled amount and invalidate only the portion of your facet that represents part content scrolled *into* view.

Drawing the contents of frames that include scroll bars is more complicated, for two reasons. First, the content must be clipped so that it won't draw over the scroll bars. Second, although the content can be scrolled, the scroll bars themselves should not move. See Drawing with Scroll Bars for more information.

Drawing Selections

You determine how selecting works and how selections are drawn inside your parts.

When a part embedded within your part is selected, the appearance you give it should be appropriate for your own selection model but it should also reflect whether the part is selected by itself or is part of a larger selection that includes your own part's intrinsic content:

- To select an embedded part alone when it is viewed in a frame, the user can perform several actions, such as activating the part and then clicking on its frame border or using a lasso tool or other selection method supported by the containing part. (Your part should support all selection methods that are appropriate for your content model. Note that when a frame is selected, its part is not active; the menus displayed are those of the containing part).

  If the part alone is selected, your part (the containing part) should draw the frame border with a standard appearance-typically a gray line, corresponding to the frame shape, with resize handles (usually eight of them) if you permit the user to resize the frame. If you allow nonrectangular frames (such as irregular polygons or circles), you are responsible for putting an appropriate number of resize handles on the frame border.

- To select an embedded part alone when it is viewed as an icon, the user places the mouse pointer over a part's icon (that is, anywhere within the active shape of the facet displaying the icon) and clicks the mouse button. Because the view type of the facet is iconic rather than framed, OpenDoc sends the mouse event to your part.

  You should, in this case, not draw the frame border of the embedded part at all. Instead you should notify the part that it is highlighted by calling the ChangeHighlight method of the embedded part's facet, specifying a highlight value of kODFullHighlight. It is then up to the embedded part to draw its highlighted appearance, either by using a highlight color or displaying the selected version of its icon.

- When an embedded part is enclosed within a range of your selected intrinsic content, its appearance should be analogous to that of the intrinsic content itself. You should draw the frame border with a selected appearance, if appropriate, and you should call the ChangeHighlight method of the embedded part's facet, so the part will know how it should highlight itself.
    - If your part highlights selections with a highlight color or inverse video (as text processors typically do), you should not draw frame borders around embedded parts within your selection, and you should set their facets' highlight value to kODFullHighlight.
    - If your part highlights selections by drawing frame borders around individual objects (as object-based drawing programs typically do), you should draw frame borders with a selected appearance around any embedded parts, and you should set their facets' highlight value to kODNoHighlight.
    - If your part highlights selections by drawing a dynamic marquee or lasso border around the exact area of the selection (as bit-map-based drawing programs typically do), you should not draw frame borders around embedded parts, and you should set their facets' highlight value to kODNoHighlight.

Your part should allow for multiple selection, so the user can select more than one frame at a time, and it should also support Select All on its own contents. Furthermore, it should allow for selection of *hot part*s -parts such as controls that normally perform an action (such as running a script) in response to a single click-without executing the action.

Drawing with Scroll Bars

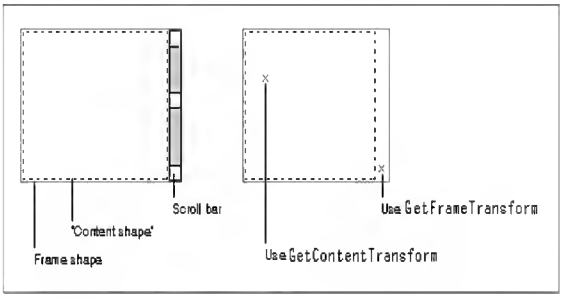
This section discusses two methods for providing scroll bars for your parts. These approaches work not only for scroll bars, but also for any nonscrolling adornments associated with a frame.

Placing Scroll Bars within your Frame

If you create scroll bars for your content inside the margins of your frame, remember that your frame shape includes the areas of the scroll bars. To draw properly, you need to account for the fact that, although your content can scroll, the scroll bars themselves should remain stationary.

Note: The method discussed in this section does not apply to a root part where the scroll bars are part of the document shell window frame control and are not within the root parts display frame.

Clipping and scrolling your content
One approach is to create an extra, private "content shape," as shown in the following figure.



You can define the content shape in terms of frame coordinates, and you can store it anywhere, although placing it in your frame's part info data is reasonable.

When drawing your part's contents-the portion within the area of the content shape-you take the current scrolled position of your content into account. You include the frame's internal transform by using the transform returned by GetContentTransform to set the origin, and you use the content shape as the facet's clip shape when drawing. If you will draw your own scroll bars, you ignore the current scrolled position of your content when drawing them. You use the transform returned by GetFrameTransform to set the default viewing transform, and you use the frame shape as the facet's clip shape. If you are using the PM scroll bar control window, you need to position the scroll bar window using the frame's external transform and set the window clip region, as described in the recipes Facet Windows and Using Embedded PM Controls within a Facet. You should call WinUpdateWindow in your Draw method, specifying the handle of the scroll bar window after drawing your content.

Clipping embedded frames
One complication of this method is that OpenDoc knows nothing of your content shape and therefore cannot clip the facets of embedded frames accordingly. OpenDoc clips all embedded facets to the area of your frame's facet, but that area includes the scroll bars as well. Thus, embedded parts can draw over the area of your scroll bars.

To avoid this problem, you must intersect the clip shapes of each embedded facet with your content shape before the facet draws itself.

Placing Scroll Bars in a Separate Frame

To avoid having to define a separate, private "content shape," you can place scroll bars or adornments in completely separate frames from your content. This method, however, requires the overhead of defining more objects, uses more memory, and may possibly affect performance negatively.

You can request one display frame for your part that encloses both the scroll bars and the content. This frame has an internal transform of identity; it does not scroll. If you are drawing your own scroll bars, draw the scroll bars directly in this frame.

You then request a second display frame for your part, a frame that can scroll and in which you draw your content. Make the second frame a subframe of the first; that is, make the nonscrolling frame the containing frame of the scrolling frame. Because both frames are display frames of the same part (yours), you must set the isSubframe parameter to true when you first create the new frame. This setting notifies OpenDoc that the frame is a subframe, so that OpenDoc draws the active frame border in the correct location-around the facet of the parent frame.

Your part's containing part, as usual, must make separate facets for both frames. Then, when a mouse event occurs in the nonscrolling frame, your part can interpret it and set the internal transform of the scrolling frame accordingly.

With this method, you do not need to take any special care to manage clip shapes of any embedded facets.

A more common use of subframes may be in creating scrollable split windows. See Drawing with Multiple Frames or Facets.

Drawing Directly to the Window

In some circumstances, you may want your part editor to draw interactively, providing feedback for user actions. For example, you may support rubber-banding or sweeping out a selection while the mouse button is held down. If so, your part can draw directly on the window of the document shell.

OpenDoc provides several ODFacet methods, analogous to those used for drawing to your own canvas, that you can use to draw directly to the window:

- The AcquireWindowContentTransform method, analogous to AcquireContentTransform, converts from your part's content coordinate space to window-canvas coordinate space.
- The AcquireWindowFrameTransform method, analogous to AcquireFrameTransform, converts from your part's frame coordinate space to window-canvas coordinate space.
- The AcquireWindowAggregateClipShape method, analogous to AcquireAggregateClipShape, transforms and intersects your part's facet clip shape with all enclosing clip shapes, resulting in a composite clip shape in window-canvas coordinate space.

Use the values returned by these methods to set up your drawing environment when you must draw directly to the window canvas. When there is no canvas in the facet hierarchy other than the window canvas, the results returned by each of these pairs of methods are the same.

Asynchronous Drawing

Your part editor may need to display its part asynchronously, rather than in response to a call to your Draw method. For example, a part that displays a clock may need to redraw the clock face exactly once every second, regardless of whether an update event has resulted in a call to redraw. Asynchronous drawing is very similar to synchronous drawing, except that you should make these minor modifications:

1. Determine which of your part's frames should be drawn. Your part can have multiple display frames, and more than one may need updating. Because your part stores its display frames privately, only you can determine which frames need to be drawn in.
2. For each frame being displayed, you must draw all facets. The frame's CreateFrameFacetIterator method returns an iterator with which you can access all the facets of the frame. (Alternatively, you can keep your own list of facets). Draw the part's contents in each of these facets, using the steps listed for synchronous drawing.

3. After drawing in a facet, call its DrawnIn method to tell it that you have drawn in it asynchronously. If the facet is on an offscreen canvas, calling DrawnIn allows the drawing to be copied into the window, because the owning part's CanvasUpdated method is called.

Offscreen Drawing

There are several situations in which you may want your part to create an offscreen canvas. For example, you may want to increase performance with double-buffering, drawing complex images offscreen before transferring them rapidly to the screen. Alternatively, you may want to perform sophisticated image manipulation, such as drawing with transparency, tinting, or using complex transfer modes.

Parts are likely to create offscreen canvases for their own facets for double-buffering, in order to draw with more efficiency and quality. Parts are likely to create offscreen canvases for their embedded parts' facets for graphic manipulation, in order to modify the imaging output of the embedded part and perhaps combine it with their own output.

Canvases are described in general in Using Canvases;

Drawing to an Offscreen Canvas

Drawing on an offscreen canvas is essentially the same as drawing on an onscreen canvas.

- You set up the environment as usual: you obtain the canvas and its pointer to the ODPlatformCanvas object, from which you obtain the presentation space. You use your facet's transform and clip information as usual.
- If you draw asynchronously on your facet's offscreen canvas, you must, as for any asynchronous drawing, call the facet's DrawnIn method to ensure proper updating of the offscreen canvas to the window canvas. If your part's facet is moved to an offscreen canvas while it is running, OpenDoc calls your part's CanvasChanged method, so that your part will know to call DrawnIn if it is drawing asynchronously.
- You perform invalidation the same way for onscreen and offscreen canvases.

Call the Invalidate or Validate methods of your display frames and their facets, to accumulate the invalid area in the update shape of the offscreen canvas.

If you want to bypass the offscreen canvas and draw directly to the document shell window, make sure you obtain the window canvas when setting up the environment. Then use the appropriate facet calls (such as AcquireWindowContentTransform instead of AcquireContentTransform) when setting the default viewing transform and clip region.

Updating an Offscreen Canvas

The part that owns an offscreen canvas is responsible for transferring its contents to the parent canvas. Only the part that created the canvas can be assumed to know how and when to transfer its contents. The canvas may have a different format from its parent (one may be a bit map and the other a display list, for example); the owner may want to transform the contents of the canvas (such as by rotation or tinting) as it transfers, or the owner may want to accumulate multiple drawing actions before transferring.

When a containing part has placed one of its embedded part's facets on an offscreen canvas, it should force the embedded part to draw before the containing part itself draws any of its own contents. This ensures that the contents of the offscreen canvas are up to date, and can be combined safely with the containing part's contents when drawn to the onscreen canvas. You can force your embedded part to draw itself by calling the embedded facet's Draw method; you can force your embedded part's own embedded parts to draw themselves by also calling your embedded facet's DrawChildren method.

If an embedded part displays asynchronously and uses Invalidate or Validate calls to modify the update shape of its offscreen canvas, the offscreen canvas calls its owning part's CanvasUpdated method to notify it of the change. The owning part can then transfer the updated content immediately to the parent canvas, or it can defer the transfer until a later time, for efficiency or for other reasons.

Drawing with Multiple Frames or Facets

OpenDoc has built-in support for multiple display frames for every part and multiple facets for every frame. This arrangement gives you great flexibility in designing the presentation of your part and of parts embedded in it. This section summarizes a few common reasons for implementing multiple frames or multiple facets.

Using Multiple Frames

Several uses for multiple frames have already been noted in this book, most of which involve the need for simultaneous display of different aspects or portions of a part's content.

- A part whose frame is opened into a part window requires a second frame as the root frame of that part window.
- A part that flows text from one frame to another naturally needs more than one frame.
- Parts that allow for multiple different presentations of their content are likely to display each kind of presentation (top view versus side view, wire-frame versus rendered, and so on) in a separate frame.
- Parts that display scroll bars, adornments, or palettes in addition to their own content might use separate frames for such items.

In general, the part that is displayed in a set of frames initiates the use of multiple frames. The containing part does, however, have ultimate control over the number and size of frames for its embedded parts.

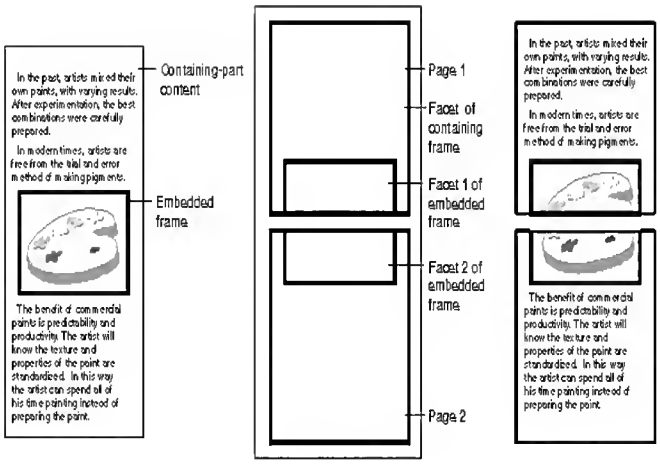
Using Multiple Facets

The use of multiple facets allows a containing part to duplicate, distort, and reposition the content displayed in its embedded frames. Normally, even with multiple facets, all of the facets of a frame display their content within the area of the frame shape, because the frame represents the basic space agreement between the embedded part and the containing part. However, the containing part always controls the facets to be applied to its embedded frames, and so the containing part can display the facets in any locations it wishes.

Drawing an Embedded Frame Across Two Display Frames

Perhaps the most common reason to use multiple facets for a frame is to show an embedded frame spanning a visual boundary between separate drawing areas in a containing frame.

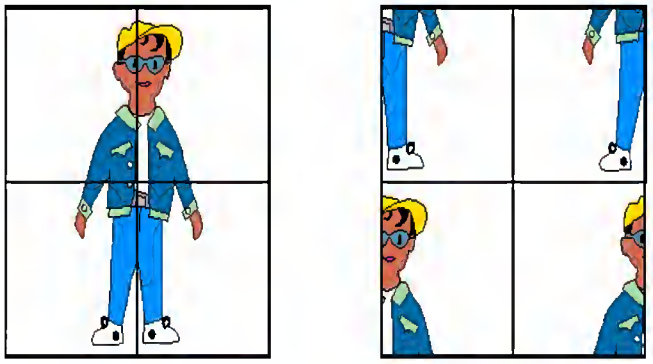
In the following figure, for example, a word-processing part uses a single display frame and facet for each of its pages. An embedded part's display frame spans the boundary between page 1 and page 2. The word-processing part then defines two facets for its embedded frame: one facet in the display for page 1, and the other in the display for page 2.



Multiple Views of an Embedded Frame

One simple use of multiple facets is for a containing part to provide a modified view (such as a magnification) in addition to the standard view of an embedded frame. You can also use multiple facets to tile the area of an embedded frame with multiple miniature images of the frame's contents.

A similar but more complex example is shown in the following figure. In this case, multiple facets are used to break up the image in an embedded frame, to allow the tiled parts to be moved, as in a sliding puzzle.



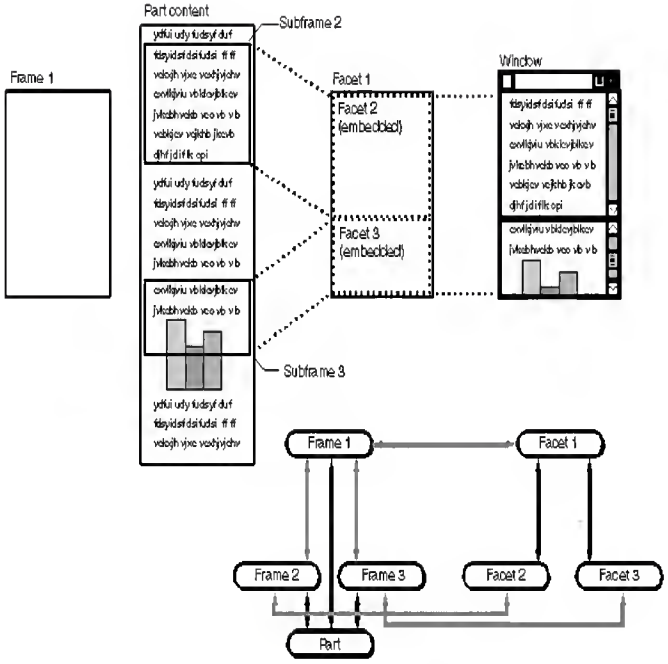
Facet shapes are sized and offset to fill frame shape

Facet shapes unchanged; external transforms modified

The clip shapes for the facets in the preceding figure are smaller than the frame shape and have offset origins so that they cover different portions of the image in the frame. Initially, as shown on the left, their external transforms have offsets that reflect the offsets in the shape definitions, and the facets just fill the area of the frame. Subsequently, as shown on the right, the containing part can cause any pair of facets to change places in the frame simply by swapping their external transforms.

Providing Split-Frame Views

One of the most useful implementations of multiple facets may be for the construction of s*plit-frame view*s , in which the user can independently view and scroll two or more portions of a single document. This use of multiple facets is somewhat more complex than the others presented here, because it also involves use of s*ubframe*s , embedded frames that display the same part as their containing frame. The following figure shows one example of how to use two subframes to implement a split-frame view.

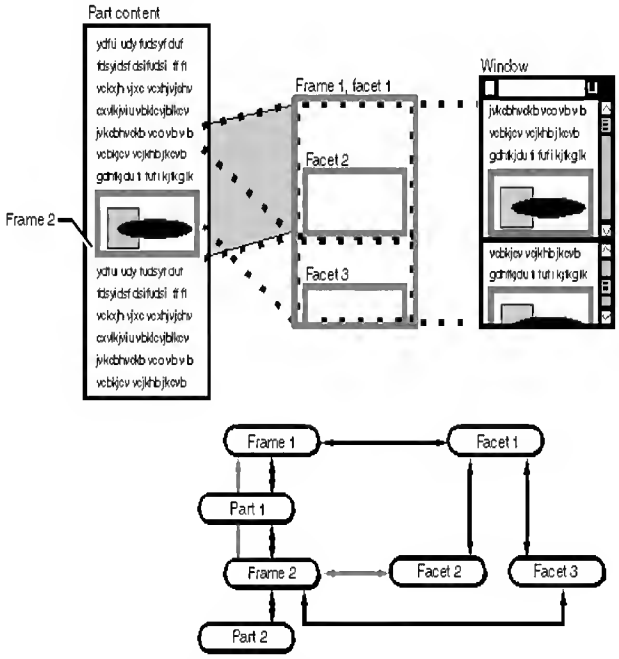


In the preceding figure, the part to be displayed has one frame (frame 1) and facet (facet 1) that represent its current overall frame size and display. The part creates frames 2 and 3 as embedded frames, specifying that the containing frame for both is frame 1, and also specifying that they are to be subframes of their containing frame. The diagram in the lower right of shows the object relationships involved, using the object-diagram conventions described in Run-Time Object Relationships.

The part now needs to negotiate for frames 2 and 3 only with itself. It creates facets for them, embedded within facet 1. The part can now position the contents of frame 2 and frame 3 independently, by changing their internal tranforms. It uses the external transforms to place facets 2 and 3 within facet 1 as shown. The two facets together fill the area of frame 1 and show different parts of the document.

Note: It is possible to achieve the same effect with only a single subframe. The facet of the parent frame can display one portion of the split view and the facet of the subframe can display the other.

You can also achieve a split-frame view with a somewhat simpler method that does not require the use of subframes. In this case, your part must perform extra work to split its content display. The method relies on multiple facets only for duplicate display of embedded parts; as the following figure shows, your part has only a single display frame and facet.



You create two facets (subfacets of your display frame's facet) for each embedded part that is visible in both halves of the split view. By your own means, you separate the display of your part's intrinsic content and, at the same time, you apply the appropriate external transforms to each embedded facet so that embedded parts appear in their proper locations in each portion of your split view. (The diagram in the lower

right of the preceding figure shows the object relationships).

Printing

To print a document, the user selects **Print** from the Document menu, and the root part displays a print dialog box (see Print document). If the user confirms the print command, the root part then sets up the environment for printing and prints the document. (When you part is the root part, it also handles the Page Setup command.)

In printing, the printer is represented by a canvas object that is separate from the screen display's canvas, although part editors in general use normal drawing calls to print their content.

Because any part can be the root part of a document, all parts should in general support the printing commands. The root part has greater responsibility in printing, but all visible embedded parts are asked to draw themselves, and they can adjust their drawing process for the presence of a static canvas. They can also directly access the platform-specific printing structure, if necessary.

OpenDoc does not replace a platform's printing interface; OpenDoc objects provide access to platform-specific structures and provide a context in which you make platform-specific calls.

This section first discusses the specific printing responsibilities of the root part, then addresses those of embedded parts, and finally notes aspects of printing that apply to all parts.

Root-Part Responsibilities

If your part is the root part, it defines the basic printing behavior of its document. It sets things up, makes the platform-specific printing calls, and performs any frame negotiation that is to occur.

Choosing a Frame for Printing

If your part's document uses the same page layout for printing as for screen display, printing is simpler. You use your part's existing display frame (the window's root frame) as the frame for printing. That way, you need not create any new frames, including embedded frames. Using your existing display frame makes most sense in situations where you do not engage in frame negotiation, because you can reuse the same layout after printing.

For greater flexibility, you may want to allow the printed version of your document to have a somewhat different layout from the onscreen version. For example, you may want to give embedded parts a chance to remove scroll bars and resize their frames or otherwise lay themselves out differently. To do that, and to retain your original layout for post-printing display, you can create a separate display frame for printing, as described next.

Printing the Document

After you have followed the steps listed in Print document, and after the user has confirmed the print dialog, your part is ready to print its document. It is assumed that you have created a platform-specific printing structure, such as a presentation space associated with a printer device context on OS/2. You should then take these general steps:

1. If you do not change layout or support frame negotiation for printing, skip this step and use your current display frame as the printing frame. Otherwise, create a separate printing frame, like this:
    - Call your draft's CreateFrame method to create a new, nonpersistent frame as a display frame for your part, to be used as the printing frame.
    - Also create new frames for each of your part's embedded parts, with the printing frame as their containing frame.
2. Create an external transform and clip shape for your printing facet. Make the clip shape equal to your page size.
3. Create a new facet for the printing frame, using the window state object's CreateFacet method. Assign the transform and clip to

the printing facet.

4. Create a platform canvas object using the facet's CreatePlatformCanvas method, and specify the printer presentation space.

5. Create a static canvas with the printing facet's CreateCanvas method. Use the SetPlatformCanvas method to assign the platform canvas object to the canvas. Create and initialize a PRINTDEST structure, and use the SetPlatformPrintJob method to assign the printing structure itself to the canvas.

6. Assign the canvas to the printing facet by calling the facet's ChangeCanvas method. Notify the printing frame of the presence of the printing facet by calling its FacetAdded method.

   (If you have not created a separate printing frame, each of your embedded parts with a facet is at this point notified of the presence of a static canvas and may attempt frame negotiation. As root part, you may permit or deny the requests).

7. Loop through all pages in your part's content area (or the page range specified in the job dialog box), and print each one. For each page, do the following:

   - Reset your printing facet's clip shape and/or transform so that it covers this page.
   - Create facets for your newly visible embedded frames, and release facets for frames no longer in view. At this point, allow frame negotiation with your part's embedded parts if you support negotiation.
   - Invalidate the printing facet's area and force a redrawing, by calling the facet's Update method.

8. When finished printing, clean up. Remove the printing facet from the printing frame and release it. Delete the platform-specific structures (such as the print job) and the platform canvas object that you have created. If you have created a printing frame, remove it by calling its Remove method.

Embedded-Part Responsibilities

Embedded parts have these few specific responsibilities in printing.

Engaging in Frame Negotiation

Your part must handle printing properly when it is an embedded part and is assigned a new display frame or facet. It should examine the facet's associated canvas to determine whether it is dynamic or static and perform frame negotiation if appropriate. On a static canvas, you might engage in a frame negotiation to make all of your part's content visible, or you might need to resize your frame to account for elimination of scroll bars or other items that are appropriate for screen display only.

The best point at which to test for the presence of a static canvas is when your facet is added or when its canvas changes, not when your Draw method is called. At drawing time it is too late to engage in frame negotiation.

Responding to GetPrintResolution

Your part may receive a call to its GetPrintResolution method as a printing job is being set up. This is its interface:

```
ODULong GetPrintResolution(in ODFrame frame);
```

Your GetPrintResolution method should first call the GetPrintResolution method of any of its own embedded parts that would be visible within the specified frame. It should then return the highest of all returned values and your own part's minimum printing resolution (in dots per inch) as the method result.

On OS/2, the print resolution for a print job cannot be set programmatically; however, a part that is initiating the print job might try to query the resolution of available printers to find one that is satisfactory.

Issues for All Parts

Your part draws to the printer canvas just as it draws to its window canvas. Your Draw method is called when the facet that displays your part is being imaged.

When your part draws to a static canvas (use the IsDynamic method of the canvas to find out), you may not want to draw adornments and interactive features such as scroll bars that are appropriate for screen display only.

On a static canvas, you may also want to perform color matching to take into account the characteristics of the device on which you are drawing.

Although all printing canvases are static, not all static canvases are for printing. The presence of a static canvas does not guarantee that a print job or job object exists. You can call the HasPlatformPrintJob method of the canvas to find out if a printing structure exists. If it does, it is available through the GetPlatformPrintJob method of the printing canvas. You might want to access the print job or job object directly to determine, for example, whether you are printing on a PostScript printer.

User Events

This is the third of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes the OpenDoc event-handling mechanism and discusses how your part editor handles user events to activate itself and to manipulate its user interface.

It is by handling user events that your part editor interacts with the user and cooperates with the user in constructing the compound documents that contain your parts.

OpenDoc distinguishes user events from semantic events, the messages related to the OpenDoc scripting extension, although user events can be converted into semantic events. Semantic events are discussed in Semantic Events and Scripting.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. For additional concepts related to your part editor's run-time environment, see OpenDoc Run-Time Features.

This chapter starts with a discussion of what user events are, and then shows how your part can

- Handle mouse events to activate itself
- Transfer various types of focus to and from itself
- Handle display-related events, for purposes such as hit-testing and scrolling

---

About Event Handling in OpenDoc

Your OpenDoc part editor is required to respond to a specific set of actions or messages from OpenDoc that constitute the majority of user interactions with your parts. In OpenDoc, these messages are called *user events* .

User events in OpenDoc include mouse clicks and keystrokes, menu commands, activation and deactivation of windows, and other events available only on some platforms. This section defines the different types of events and discusses how your part editor's HandleEvent method handles them.

How User Events Are Handled

Certain user events are handled by the dispatcher. The dispatcher accepts the user events from the underlying operating system through the document shell and dispatches them to the part editors. The dispatching system is modular and can be extended to handle new classes of events by adding a dispatch module. Each dispatch module is responsible for deciding which frame or part should be asked to handle each event that the module deals with.

When the OpenDoc dispatcher receives an event intended for your part, it locates a dispatch module for the event. The dispatch module in turn calls your part's HandleEvent method. (Your part's Draw method can be called indirectly because of an event, through the Update method of a window or facet).

Your part becomes the target for a specific type of user event (other than geometry-based events) by obtaining the *focus* for that event. The OpenDoc arbitrator keeps track of which part owns which foci by consulting a focus module, which tracks, for example, which part is currently active and therefore should receive keyboard events. Foci and focus modules are described further in Focus Types.

Geometry-based events, such as mouse clicks, are generally dispatched to the parts within which they occur regardless of which part currently has the selection focus-that is, regardless of which part is currently active. This permits inside-out activation to occur.

Your part receives some of the information about its events in the ODEventData structure. This structure is described in *OpenDoc Programming Reference* .

Other information about the event is passed to your part in an OpenDoc-defined *event-info structure* :

For any geometry-based event (mouse event) that is passed to your part, the coordinates of the event are passed in the *where* field of the structure. For events in embedded frames that are passed to your part, the frame and facet within which the event occurred are passed in the *embeddedFrame* and *embeddedFacet* fields.

Types of User Events

Your parts must, in general, handle the following types of user events:

- Mouse events
- Mouse events in embedded frames
- Keyboard events
- Menu events
- Window events
- Activate events
- Update events
- Null events
- Other events

The following sections describe how you handle each of these kinds of events.

Mouse Events

When the user presses or releases the mouse button while the mouse pointer is in the content area of an OpenDoc window, the dispatcher finds the correct part to handle the mouse event by traversing the hierarchy of facets in the window.

The dispatcher searches depth-first (trying the most deeply embedded facets first) and front-to-back (trying the front-most of sibling facets first). The dispatcher sends the event to the editor of the first-that is, most deeply embedded and front-most frame it finds whose active shape contains the pointer position. In this way, the smallest enclosing frame surrounding the pointer location receives the mouse event, preserving the OpenDoc inside-out activation model. None of the part editors of any containing parts in that frame's embedding hierarchy are involved in handling the event.

When the user presses a mouse button while the pointer is within a facet of your part, the dispatcher calls your part's HandleEvent method, passing it an event containing the mouse message.

When the user releases the mouse button while the pointer is within your facet, the dispatcher again calls your part's HandleEvent method, passing the mouse message.

The event-dispatching code does not itself activate and deactivate the relevant parts; it is up to the editor of the part receiving the mouse event to decide whether to activate itself or not. See Mouse Events, Activation, and Dragging for information on how your part should handle mouse-down and mouse-up events within its facets for the purposes of part activation, window activation, and drag-and-drop.

The dispatcher also tracks pointer position at all times when the mouse button is not pressed.

- When the pointer enters a facet of your part, moves within your facet or leaves your facet, the dispatcher calls your part's HandleEvent method, passing a WM_MOUSEMOVE event.

You can use these events to, for example, change the cursor appearance. See Handling Mouse Events.

In some situations, OpenDoc redirects or changes mouse events:

- If the user holds down the Shift key, Alt key or Ctrl key while pressing or releasing the mouse button, the dispatcher instead sends the event to the frame with the selection focus. This allows users to extend selections by Shift-clicking, Ctrl-clicking or Alt-Clicking.
- If the mouse event is in the title bar or re-size box of a window, the dispatcher converts it to a window event.
- If the mouse event is in the menu bar, the dispatcher converts it to a menu event.

- If a frame has the modal focus (see Acquiring and Relinquishing the Modal Focus), mouse events that occur outside of its border are sent to it. However, mouse events that occur within frames embedded within the frame with the modal focus are sent to the embedded frames, as expected.

- If a frame has the mouse focus (see Mouse-Up Feedback while Drawing), it receives all mouse-within events that occur, regardless of their location.

Mouse clicks within controls associated with your part can be handled in a number of ways, as discussed in Controls.

Mouse Events in Embedded Frames

If your part contains embedded frames, the dispatcher can also send you special mouse events that occur within and on the borders of the embedded frames' facets.

The following events occur when your part's frame is the active frame and the embedded frame is selected, bundled, or in icon view type, or if the user Shift-clicks or Ctrl-clicks or Alt-clicks in the embedded frame to extend a selection:

- When the user presses a mouse button while the pointer is within a facet of the embedded frame, the dispatcher calls your part's HandleEvent method, passing it an event type of kODEvtMouseDownEmbedded.

- When the user releases a mouse button while the pointer is within the embedded facet, the dispatcher again calls your part's HandleEvent method, this time passing it an event type of kODEvtMouseUpEmbedded.

  There is one exception to this. If the embedded frame is selected (rather than bundled or in icon view type), and if the mouse-up is *not* part of a Shift-click, Ctrl-click or Alt-click, your part does *not* receive the mouse-up event. In such a case, if the user initiates a drag the mouse-up event is converted to a drop; if the user does not initiate a drag, the mouse-up event is sent to the embedded frame so that the embedded part can activate itself. See Mouse Events, Activation, and Dragging for more information, including how to handle events in a background process.

The following event occurs when the frame embedded within your part is the active frame:

- When the user presses the mouse button while the pointer is within the active border of a facet of the embedded frame, the dispatcher calls your part's HandleEvent method, passing it an event type of kODEvtMouseDownBorder.

These events allow your part to activate itself and select or drag the embedded part.

Keystroke Events

When the user presses a key on the keyboard and your part has the keyboard focus, the dispatcher calls your part's HandleEvent method, passing it an event type of kODEvtKeyDown. When the user releases the key, the dispatcher again calls your part's HandleEvent method, this time passing it an event type of kODEvtKeyUp.

Exceptions to this convention include keystrokes that are keyboard equivalents to menu commands when the menu has the focus, which go to your part as menu events, and keystroke events involving the **Page Up**, **Page Down**, **Home** and **End** keys, which go to the frame-if any-that has the scrolling focus.

Menu Events

If the user presses the mouse button when the mouse pointer is within the menu bar, or if the user enters a keyboard equivalent to that action, the OpenDoc converts the mouse-down or keystroke event into a menu event of type kODEvtMenu and calls the HandleEvent method of the part with the menu focus.

On the OS/2 platform, the *message* field of the event structure passed to HandleEvent contains WM_COMMAND. Processing of the WM_COMMAND message for menu events is handled the same as for a PM application window procedure.

For a description of the HandleEvent method, see the *OpenDoc Programming Reference* . Handling individual menu commands is discussed in Menus.

Window Events

If the user presses the mouse button while the pointer is within a non-content position of the window (such as the close icon in the window), or if the user enters a keyboard equivalent to that action, OpenDoc converts the mouse-down or keystroke event into an event of type kODEvtWindow and the document shell handles the event.

How to handle window events when your part is the root part is described in Handling Window Events.

Activate Events

On platforms that support activation and deactivation of windows, OpenDoc sends activate events and deactivate events, of type kODEvtActivate, to each facet in the window when the window changes state.

If the user clicks in the title bar of an inactive window, OpenDoc activates the window and brings it to the front. If the user clicks in the content area of an inactive window, the part at the click location brings the window to the front. Either way, when an active window becomes inactive because another window has become active, each facet of each part displayed in the window being deactivated receives a deactivate event, and each facet of each part displayed in the window being activated receives an activate event.

Your part's HandleEvent method can use these events to store and retrieve information with which it can decide whether or not to activate itself when its window becomes active; see Handling Activate Events for an explanation.

Update Events

To support redrawing of previously invalidated areas of a window, frame, or facet, OpenDoc handles update events and calls the Draw methods of the appropriate parts.

Update events are themselves triggered by the existence of invalid areas, created through changes to the content of parts, the activation of windows, or the removal of obscuring objects such as floating windows.

OpenDoc does not pass update events to your HandleEvent method. When an update event occurs that involves a facet of your part, the dispatcher calls the Update method of the window, which results in a call to your Draw method. For more information, see Invalidating and Updating.

Propagating Events

A containing part can set a flag in an embedded frame that allows the containing part to receive events not handled by the embedded frame. If your part contains embedded frames with that flag set, your HandleEvent method receives the events originally sent to them. OpenDoc sets the *propagated* field in the EventInfo structure to true when it passes a propagated event to your part.

Whenever you add a facet to a frame, you can check the state of the flag by calling the frame's DoesPropagateEvents method. You can then set or clear the flag by calling the frame's SetPropagateEvents method.

This is a specialized feature of OpenDoc, not likely to be used by most part editors. You might use it to manipulate embedded selections; for example, you could use tab-key events to allow the user to tab between embedded parts that do not themselves support tabbing or otherwise handle tab-key events.

If you do not set the event-propagating flag for any of your embedded frames, your HandleEvent method receives only those embedded-frame events described in Mouse Events in Embedded Frames.

HandleEvent Method of your Part Editor

The dispatcher calls your part's HandleEvent method to pass it a user event meant for your part. This is the interface to the method:

```
ODBoolean HandleEvent(inout ODEventData event,
                      in ODFrame frame,
                      in ODFacet facet),
                      inout ODEventInfo eventInfo);
```

Your implementation of HandleEvent might be similar to this:

1. Initialize the result flag to kODFalse.
2. Obtain the part info data from the frame or facet to which the event was directed, if you have stored relevant information there.
3. Execute a switch statement with one case for each event type. Pass execution to the appropriate handler for each event type. Set the result flag to kODTrue if an individual handler handles the event.
4. Return the result flag as the method result.

Your individual event handlers should function as described in other sections of this chapter. Each should return kODTrue to HandleEvent if it handles an event, and kODFalse if it does not.

Mouse Events, Activation, and Dragging

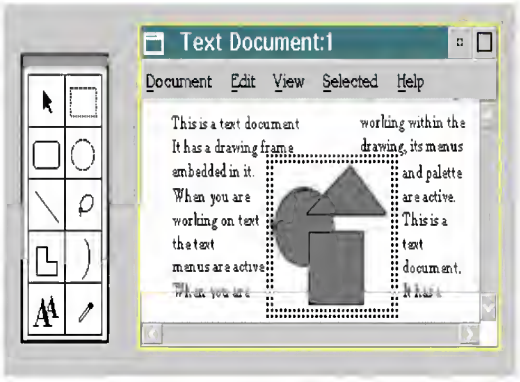
This section discusses how your part should respond to mouse-down, mouse-up, and activate events in order to activate your part or your window, initiate a dragging operation, or create a selection. Your HandleEvent method (see HandleEvent Method of your Part Editor) should dispatch events to routines that perform the actions described here. For other information about creating and modifying selections, see Mouse-Down Tracking and Selection.

How Part Activation Happens

A part's frame activates itself in these situations:

- When it receives a mouse event within its content area
- When its window first opens and the part has stored information specifying that the frame should be active on opening
- When its window becomes active and the part has stored information specifying that the frame should be active upon window activation
- When data is dropped on the part as a result of a drag-and-drop operation
- When it has another reason for becoming active (such as the need to display a selection, such as a link source, to the user)

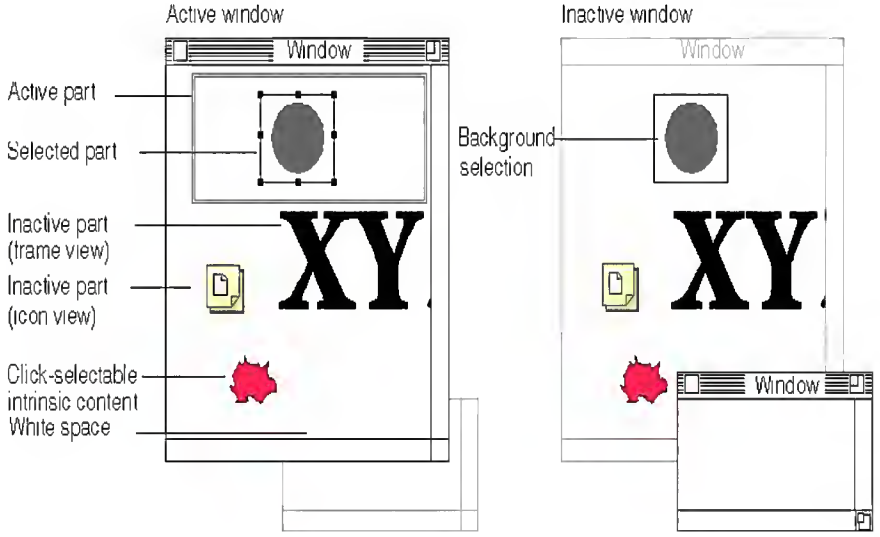
The part activates itself by acquiring the selection focus (see Focus Types) for one of its display frames. The part is then responsible for displaying its contents appropriately (such as by restoring the highlighting of a selection) and for providing any menus, controls, palettes, floating windows, or other auxiliary items that are part of its active state. The part must also obtain ownership of any other foci that it needs. OpenDoc draws the active frame border around the frame with the selection focus. the following figure illustrates some of the visual changes caused by activation.



In the preceding figure, the graphics part is active. OpenDoc draws the active frame border around its frame, it has a highlighted selection, and it displays Font, Size, and Style menus.

Typically, one part is deactivated when another is activated. A part is expected to deactivate itself when it receives a request to relinquish the selection focus, plus, possibly, other foci. The deactivating part is responsible for clearing its selections-an active part in the active window should not maintain a background selection-and for removing any menus and other items that were part of its active state. OpenDoc removes the active frame border from around the (now inactive) part and draws it around the part that currently has the selection focus.

Mouse events that cause part activation can also cause window activation and can lead to dragging of part content. The following figure illustrates some of the elements you must consider in handling mouse-down events for these purposes. As the following sections explain, your part should handle mouse events according to whether your part is active or inactive, whether it is in an active or inactive window, and whether the event location corresponds to one of the elements- such as the active frame border or an embedded part's icon-that are shown in the following figure.



Note these points from the previous figure:

- Embedded parts can be represented by either frames or icons. Frames can be active or inactive, and inactive frames can be either selected or unselected. (Not shown in the figure is that inactive frames can also be bundled).
- There can be only one active frame in the active window, and no active frame in an inactive window (except that a dialog box can appear in front of a window that contains the active part).
- All selected items must be within the active frame.
- A selection in the active window becomes a background selection if the window becomes inactive.
- A selection (either foreground or background) can consist of either intrinsic content or an embedded frame or icon, or a combination of both.
- "Click-selectable intrinsic content" in the figure means any item of intrinsic content that can be selected by a single mouse click, such as a graphic object in a draw part.
- "White space" in the figure means any location in the content area of a part that is neither click-selectable intrinsic content nor an embedded part frame or icon. It includes empty space, but also includes content (such as text or painting) that requires a sweeping gesture, rather than a single click, for selection.

- An inactive window may be in the current process (if, for example, it is a part window of the active document) or in a background process (if, for example, it is a window of another document).

The event-handling guidelines given here are designed to provide the user with maximum consistency and flexibility when activating parts and when dragging selections.

Handling Mouse Events

On the OS/2 platform, mouse events are sent to your part's HandleEvent as normal PM mouse messages. The *flags* field of the EventInfo structure contains information about where the mouse event occurred. It will be either kODPropagated, kODInBorder or kODInEmbedded.

Handling Activate Events

As noted in Activate Events, your part receives an activate event for each of its facets when your window becomes active, and a deactivate event when your window becomes inactive.

When a window becomes inactive, the part that holds the selection focus is likely to relinquish that focus (on request by another part) but can still maintain, as a background selection, any selection it had been displaying. Conversely, when a window becomes active, the part that had held the selection focus when the window became inactive normally regains the selection focus. That part may or may not be the part that actually activated the window.

This part-activation convention requires that your part respond to activate and deactivate events by recording or retrieving stored information on its focus transfers; see On Window Activation for more information.

Activate events in a window go first to the most deeply embedded facets and last to the root facet. This dispatch order gives the root part the opportunity to obtain the selection focus if no embedded part has done so. It also allows the root part to override any activation actions taken by embedded parts.

Focus Transfer

Part activation is the process of making a part and frame ready for editing. As noted previously, activation typically occurs when the user clicks the mouse button when the pointer is within a frame that is not currently active, but also happens when a window opens, when a window is activated, and as a result of drag-and-drop operations.

In the OpenDoc model of part activation, part editors use the concept of *focus* to activate and deactivate themselves (rather than being activated and deactivated by OpenDoc) and to arbitrate the transfer of several types of shared resources among themselves.

Focus Types

A part makes itself the recipient of a certain type of user event or other action by obtaining the focus for it. A *focus* is a designation of ownership of a given shared resource, feature, or event type; for example, the frame that owns the keystroke focus receives all keystroke events until it passes ownership of the keystroke focus to another frame.

Focus types are defined as ISO strings, and the standard set defined by OpenDoc includes those in the following table.

Constant	ISO String	Description
kODKeyFocus	"Key"	Keystroke events are sent to the frame with this focus.
kODMenuFocus	"Menu"	Menu events are sent to the frame with this focus.

kODSelectionFocus	"Selection"	Shift-click, Ctrl-click and Alt-click mouse events are sent to the frame with this focus. OpenDoc draws the active frame border around all facets of this frame.
kODModalFocus	"Modal"	The frame that owns this focus is notifying other frames that it is the only current modal frame.
kODScrollingFocus	"Scrolling"	Scrolling-specific keystroke events (such as Page Up and Page Down) are sent to the frame with this focus.
kODClipboardFocus	"Clipboard"	The frame that owns this focus has access to the clipboard.
kODMouseFocus	"Mouse"	The frame that owns this focus receives all mouse-within events whenever the pointer moves, and all mouse-down and mouse-up events, regardless of which facet the pointer is within.
kODStatusLineFocus	"StatusLine"	The frame that owns this focus has access to the status line window.

To obtain event foci, part editors request them by name from the arbitrator. (You obtain access to the arbitrator by calling the GetArbitrator method of the session object).

You need to convert focus names into tokens before using them in any method calls. You call the Tokenize method of the session object to convert ISO strings to tokens.

Foci may be manipulated singly or in groups called focus sets. A focus set is an OpenDoc object (of class ODFocusSet) listing a group of foci that a part editor wants to obtain or release as a group.

Foci are owned by frames. In general, mouse events anywhere within the content area of an OpenDoc window always go to the most deeply embedded frame that encloses the click point. However, Shift-click, Ctrl-click or Alt-click events, regardless of their location, are sent to the frame with the selection focus to allow for extending selections.

OpenDoc does not require that the same frame own the selection focus, keystroke focus, and menu focus, although this is most often the case. Nor does OpenDoc require that the selection focus be in an active window, although this is usually the case.

In most cases, when a frame is activated, the part editor for that frame requests the selection focus, keystroke focus, and menu focus. A frame with scroll bars might also request the scrolling focus. Your part editor might create a focus set ahead of time, perhaps during part initialization, that includes the tokenized names of the foci that your part expects to request when it becomes active. You use the arbitrator's CreateFocusSet method to create the focus set.

A simple part, such as a small text-entry field in a dialog box, might request only the selection focus and keystroke focus on receiving a mouse-up event within its frame area. An even simpler part, such as a button, might not even request the selection focus. It might simply track the mouse until the button is released and then run a script, never having changed the menu bar, put up palettes or rulers, or become active.

Your part editor can define additional focus types as needed. You can define other kinds of focus, perhaps to handle other kinds of user events (such as input from new kinds of devices). To create a new kind of focus, you need to create a new kind of focus module, the OpenDoc object that the arbitrator uses to determine focus ownership. Semantic Events and Scripting describes how to use focus modules to extend OpenDoc's focus management.

Foci may be exclusive or nonexclusive. All of the standard foci defined by OpenDoc are exclusive, meaning that only one frame at a time can own a focus. But if you create a new kind of focus, you can make it nonexclusive, meaning that several frames could share ownership of it.

Arbitrating Focus Transfers

This section discusses how to request or relinquish foci to activate or deactivate your frames.

Requesting Foci

A part can request, for one of its frames, ownership of a single focus or a set of foci. You request a focus by calling the arbitrator's RequestFocus method; you request a focus set by calling the arbitrator's RequestFocusSet method. If the request succeeds, your part's frame obtains the focus or focus set.

The arbitrator's RequestFocus and RequestFocusSet methods perform a two-stage transaction in transferring a focus or focus set:

1. The arbitrator first asks the current owning frame of each focus if it is willing to relinquish the focus, by calling the BeginRelinquishFocus method of the frame's part.

2. If any owner of the focus is unwilling to relinquish it, the arbitrator cancels the request by calling each part's AbortRelinquishFocus method. In this case, RequestFocus or RequestFocusSet returns false.

3. If all focus owners are willing to relinquish, the arbitrator calls each part's CommitRelinquishFocus method. In this case, RequestFocus or RequestFocusSet returns true.

Relinquishing Foci

A part can relinquish foci either on request or when a change to its state (such as the closure of its frame or a completion of a method) warrants it. An active part might unilaterally relinquish certain foci (such as the clipboard focus) as soon as it is finished handling an event, but it might not relinquish other foci (such as the selection focus) until another part asks for them. Nevertheless, most parts willingly relinquish the common foci when asked.

Relinquishing foci on request is a two-step process, because multiple foci requested as a focus set must all be provided to the requestor simultaneously; if one is not available, none need be relinquished. Your part editor participates in the process through calls to its BeginRelinquishFocus, CommitRelinquishFocus, and AbortRelinquishFocus methods.

1. In your BeginRelinquishFocus method, you need do nothing other than return kODTrue or kODFalse, based on the type of focus and the identities of the frames (current and proposed focus owners) passed to you. In most cases, you can simply return kODTrue, unless your part is displaying a dialog box and another part is requesting the modal focus. In that case, because you do not want to yield the modal focus until your dialog box window closes, you return kODFalse. See Acquiring and Relinquishing the Modal Focus for more information.

2. Your part's CommitRelinquishFocus method, verifies that you have actually relinquished the focus type you responded to in BeginRelinquishFocus. The method should take appropriate action, such as removing menus or palettes, disabling menu items, removing highlighting, and performing whatever other tasks are part of losing that type of focus. Remember that the focus may possibly be moving from one frame to another of your part, so the exact actions can vary.

3. If, after your part responds with kODTrue to BeginRelinquishFocus, the focus is actually not transferred from your frame, OpenDoc calls your part's AbortRelinquishFocus method. If your part has done anything more than return the Boolean result in BeginRelinquishFocus, it can undo those effects in the AbortRelinquishFocus method.

4. If your part is one of several focus owners called to relinquish the foci of a focus set, and if you return kODFalse to BeginRelinquishFocus, your CommitRelinquishFocus method is not called (because you chose not to give up the focus). However, your AbortRelinquishFocus method is still called (because all owners of a focus set are notified if any one refuses to relinquish the focus).

Your part does not relinquish its focus on request only. For example, in your part's DisplayFrameClosed and DisplayFrameRemoved methods, you should include a call to the arbitrator's RelinquishFocus or RelinquishFocusSet method to unilaterally relinquish any foci owned by the frame that you are closing. When your part closes, its ReleaseAll method should likewise relinquish all of its foci. When your part finishes displaying a modal dialog box, it should relinquish or transfer the modal focus; when your part finishes accessing the clipboard, it should relinquish the clipboard focus.

Transferring Focus without Negotiation

There are some situations in which the normal process of requesting and relinquishing foci is not used. Another piece of software interrupts your part's execution, and your part loses a focus without being given a chance to relinquish it, or gains focus without having asked for it. To handle those situations, your part editor must implement versions of the methods FocusAcquired and FocusLost. The arbitrator calls these methods when your part has just acquired, or just lost, a specified focus without having negotiated the transaction.

For example, a containing part, to support keyboard navigation, might call FocusAcquired in turn on each of its embedded parts as the user makes successive keystrokes. Or, if a custom input device with its own focus type were in use and then became detached, the part using the device might receive a call to its FocusLost method.

These are the interfaces to FocusAcquired and FocusLost:

```
void FocusAcquired(in ODTypeToken focus,
                   in ODFrame ownerFrame);

void FocusLost(in ODTypeToken focus,
               in ODFrame ownerFrame);
```

Your FocusAcquired and FocusLost methods should perform any actions your part editor deems appropriate in response to having just acquired or lost a focus.

The arbitrator's methods TransferFocus and TransferFocusSet allow you to initiate a transfer of focus ownership without negotiation. A part can use these calls to transfer focus among parts and frames that it controls directly. For example, in a modal dialog box consisting of several parts, these methods can be used to transfer a focus from the outer part (the dialog box) directly to an inner part (such as a text field) and back.

When focus is transferred with TransferFocus or TransferFocusSet, the arbitrator calls the FocusAcquired method of the new frame's part and the FocusLost method of the previous frame's part-except that, when the frame performing the transfer (the frame representing the part that calls TransferFocus) is the frame receiving or losing the focus, its FocusAcquired or FocusLost method is not called.

Calling your Own FocusAcquired and FocusLost

It might seem natural to call your own FocusAcquired method when your request for foci succeeds, or to call your own FocusLost method from your own CommitRelinquishFocus method. A better practice, however, is to have related methods call a shared private method, so that you maintain a clear separation between public and private interfaces.

Recording Focus Transfers

Different frames may need different sets foci when activated. Selection focus, keystroke focus, and menu focus are commonly needed together. However, a frame with scroll bars might also need the scrolling focus, and a frame for a modeless dialog box might not even want the selection focus.

OpenDoc does not save or restore focus assignments. Therefore, during deactivation of windows and frames, and during closing of windows, you can record the state of focus ownership in order to restore it at a later activation or reopening. Your display frame's part info is an appropriate place to keep that information. Your part's initialization method might create a focus set with those foci, to use whenever your display frames become active.

On Frame Activation

When a previously inactive frame in a window becomes active, the part editors involved should-besides negotiating the focus transfer-record the gain or loss of selection focus for the respective frames. If you maintain that information, your activation and deactivation routines can check the state and exit quickly if not needed.

- If you are activating your part's frame, you might record, in a Boolean flag with a name such as fHasRequiredFoci in the frame's

part info data, the fact that the frame has the selection focus. You can perform this action in your FocusAcquired method, after its call to the arbitrator's RequestFocusSet method succeeds.

- If you are deactivating your part's frame, you might set the fHasRequiredFoci flag in the frame's part info data to false. You can perform this action in your FocusLost method and/or your CommitRelinquishFocus method.

On Window Activation

As mentioned in Activate Events, all parts displayed in a window receive an activate event when the window becomes active, and a deactivate event when the window becomes inactive.

When an active facet of a frame of your part becomes inactive through window deactivation, your part's HandleEvent method can-upon receiving the deactivate event-store a flag in the facet's part info field to note that the facet was active before window deactivation. Your part then can also maintain, as a background selection, any selection it had been displaying.

Conversely, when a facet of a frame of your part receives an activate event because of window activation, your part's HandleEvent method can examine the state of the flag in the part info field to decide whether or not it was the active part when the window became inactive. If so, it should request the selection focus, reset the flag, and convert any background selection it may have maintained into a foreground selection.

On Closing and Reopening Documents

Normally, the root part of a newly opened window should activate itself as a matter of course. However, if an embedded had the selection focus when the window closed, the root part can-if it chooses to-allow the embedded part to recapture that focus when the window reopens.

When the state of a window is saved in a document and the document is subsequently reopened, the root part recreates the window. If you want to restore the selection-focus state of your part (plus perhaps the selection itself), you can save the selection and the state of the selection-focus flag in your frame's part info data when the window is closed, and restore them when the window is opened (when your part's DisplayFrameConnected method is called).

If your part is the root part in this situation, you can either allow the embedded part's request for selection focus at this time, or you can acquire the selection focus yourself, when your own DisplayFrameAdded or DisplayFrameConnected method is called. (The root part is called last).

Display-Related Events

Your part editor needs to respond to events that select or change the position or visibility of your intrinsic content, by highlighting or preparing to move or redraw the content. This section discusses general event-handling issues related to drawing and highlighting. Display concepts are described more completely in Frames and Facets.

Just as with intrinsic content, your part editor is responsible for knowing what embedded frames look like and when they become visible or invisible. When update events or events related to scrolling or editing cause an embedded frame to become visible, your part is responsible for creating facets for those frames. If events cause an embedded facet to move, your part must modify the facet's external transform to reflect the move. If the embedded facet moves so as to become no longer visible, your part is responsible for deleting the embedded facet from its containing facet (or marking it purge-able).

Hit-Testing

Hit-testing is the interpretation of mouse events in terms of content elements within a frame. For example, when the user clicks the mouse somewhere within your part's content area to select an item or to set an insertion point, you must use the click location to decide which item has been selected or which characters correspond to the insertion point. You can then highlight the proper item or draw the text caret at the proper location.

Coordinate conversion is necessary to assign content locations to hit-testing events; in that sense, hit-testing is the inverse of drawing. Coordinate conversion and the application of transforms during drawing is discussed in Transforms and Coordinate Spaces.

If a mouse click occurs in a facet of your embedded part's frame, OpenDoc applies the inverse of all external transforms from the canvas facet to your facet, passing to you the mouse event in your part's frame coordinates. These coordinates are stored in the *ODPoint* field of the EventInfo structure. It is then your responsibility to apply the inverse of your own internal transform to the event location to convert it to the content coordinates of your part.

If the entire content of your part's frame is scrolled, the application of your frame's internal transform yields the correct location for everything within the frame. If, however, your part has drawn scroll bars or other nonscrolling items within the frame, it is important not to apply the internal transform to events over those items. See Scrolling for more information.

Invalidating and Updating

When keystroke or mouse events involved with editing have changed the visible content of your part, you should invalidate the affected areas (by calling your frame's Invalidate method), so that an update event will be generated to force a redraw. If the same presentation of your part is displayed in more than one frame (see Synchronizing Display Frames), you are responsible for invalidating those frames as well, if necessary.

Sometimes, a portion of a window needs to be redrawn because it has been invalidated by actions taken by the documents' parts or by the system (as when a window is uncovered). In this case, the document shell receives notification of that fact and passes an update event to the dispatcher, which calls the Update method of the window. The window in turn calls the Update method its root facet.

The root facet's Update method examines all of the root facet's embedded facets (and their embedded facets, and so on) and marks each one that intersects the area that must be updated. Then, each facet that needs to be redrawn calls the Draw method of its frame's part, passing it the appropriate shape that represents its portion of the area to be updated.

Your part can, if desired, modify the order in which embedded parts draw their content, by making explicit calls to the Draw, DrawChildren, and DrawChildrenAlways methods of an embedded facet. See Draw Method of your Part Editor and Asynchronous Drawing for more information.

Scrolling

The scrolled position of a part's contents within a frame is controlled by the frame's internal transform. (The transform object can also support scaling, translation, rotation, and perspective transformations of a frame's contents. See the figure on transforms in Transforms and Coordinate Spaces for an example.

In general, the user specifies the amount of scrolling to apply to a frame. The part editor then modifies the offset specified in the frame's internal transform. (Depending on the new scrolled position of the part's contents, the part editor might also have to add or remove facets for embedded frames that have become visible or obscured by the scrolling).

There are a number of ways for your part editor to support scrolling:

- It can create scroll bars for its content, placing them at the margins of its display frames.
- It can create scroll bars, sliders, or other kinds of controls and place them outside of its frame, as separate frames, or as elements in a floating window.
- It can support auto-scrolling, by tracking the mouse pointer when the user moves it beyond your part's frame, while holding down the mouse button.

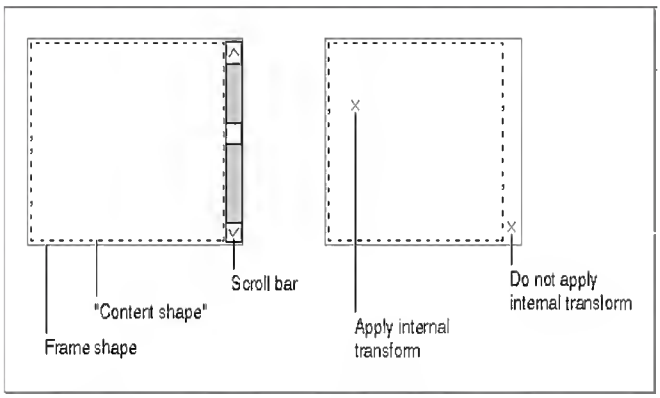
Each of these methods requires different event handling by your part to achieve the same result: a modified internal transform for the frame. Your part then draws the scrolled display as described in Scrolling your Part in a Frame.

Event-Handling in Scroll Bars within your Frame

You can create scroll bars for your content inside the margins of your frame. One approach to handling events in those scroll bars is to create an extra, private "content shape" within your frame. The content shape is the same as the frame shape, except that it does not include the areas of the scroll bars.

When interpreting mouse events within your frame, it is important that the current scrolled position of your content be taken into account for content editing but not for scroll-bar manipulation. Thus, for points that fall within your content shape, you apply the inverse of the frame's

internal transform to mouse events, whereas outside of it (in the scroll bar area) you do not. See the following figure.



Mouse events within the scroll bar area specify how to set the frame's internal transform; based on the transform's new value, you then redraw your part's content (and the scroll bar slider). Mouse events within the area of the content shape specify how to select or edit the appropriately scrolled content.

Event Handling in Scroll Bars in Separate Frames

If you place scroll bars or adornments in completely separate frames from your content, (see Placing Scroll Bars in a Separate Frame), you avoid having to define a separate content shape. Your part has one display frame that encloses both the scroll bars and the content; you draw the scroll bars directly in this frame, and you draw the content in a sub-frame. Only the sub-frame's internal transform changes.

When a mouse event occurs in the nonscrolling frame, your part interprets it and sets the internal transform of the scrolling sub-frame accordingly.

Mouse-Down Tracking and Selection

A selection exists only in the context of a particular part. The boundary of a selection cannot span part boundaries; all of its margins must be in the same part. However, a selection can include any number of embedded frames (which themselves can be considered to include any number of more deeply embedded frames).

Selections are typically created by mouse events, or keyboard-modified mouse events, within an already active frame or a frame that has been made potentially active by an initial mouse-down event within its area. When a mouse-down event occurs, OpenDoc passes the pointer location, in the frame coordinates of the most deeply imbedded frame enclosing the event location, to the part displayed in that frame. If a drag-selection occurs (that is, if a drag-and-drop operation does not occur), the subsequent mouse-up event location is passed in the same coordinates to the same part. Thus, the active frame is not changed simply because the cursor passes across a frame boundary while making a selection.

The selection actions your part takes during mouse-down tracking depend on your content model. You should provide feedback to the user while the dragging is in progress.

Your part editor should support extending selections through Shift-clicking, Ctrl-clicking and Alt-clicking. Note that, because selection is possible only within one part at a time, a part editor can extend a selection outside of its display frame boundary only to items within another of its own display frames.

If your part is a containing part and allows the user to drag selected content, including embedded frames, dragging may lead to frame negotiation. If an embedded frame is clipped by the edge of a page, for example, you may need to change its size.

Selected frames should be highlighted appropriately, as described in Drawing Selections.

Mouse-Up Tracking

OpenDoc calls your part's HandleEvent method whenever the mouse pointer enters, moves within, or leaves a facet of a display frame of your

part and the mouse button is up (not pressed). Your part can use this method call to display a custom cursor while the pointer is within your part's frames.

Your part is responsible for providing the appropriate cursor appearances as defined by OpenDoc.

Mouse-Up Feedback while Drawing

In certain modal situations, such as when drawing polygons or connected line segments, the user typically draws by clicking the mouse button once for each successive vertex or joint. (The user might complete the polygon and exit the mode by double-clicking or by clicking in the menu bar or another window). During the drawing operation, while the mouse button is up, the part editor must provide feedback to the user, showing a potential line segment extending from the last vertex to the current mouse position.

Your part can support this kind of drawing feedback by requesting the mouse focus (kODMouseFocus) when the user clicks to make the first vertex after selecting the appropriate drawing mode (perhaps from a tool palette that you maintain). Your part then receives events when the mouse moves, regardless of which facet the cursor travels over; the facet passed to your HandleEvent method is the one in which the initial mouse-down event occurred. Receiving these events allows you to provide visual feedback regardless of where the user moves the mouse pointer.

Your part also receives all mouse-down and mouse-up events until it relinquishes the focus. In addition, as long as your part holds the mouse focus, OpenDoc sends no mouse events to any facet.

When the user completes the drawing operation, relinquish the mouse focus.

If the user clicks on the desktop or on another document's window, your part receives a suspend event and should exit the mode (relinquish the mouse focus). Also, if the user clicks in the menu bar, you must relinquish the mouse focus and redispatch the event if you want the menu to appear.

Windows and Menus

This is the fourth of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes how your part editor can present and manipulate some of the major elements of its user interface.

This chapter is a continuation of the previous chapter: it discusses programming issues involved with your part's user interface.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. For additional concepts related to your part editor's run-time environment, see OpenDoc Run-Time Features.

This chapter discusses the following topics:

- Windows
- Dialog boxes
- Controls
- Menus
- Undo

Windows

Windows are platform-specific data structures through which documents display themselves. This section discusses how to use OpenDoc window objects (which are basically wrappers for those structures) regardless of which platform you are developing for.

Creating and Using Windows

The OpenDoc class ODWindow is a wrapper for a pointer to a platform-specific window structure. For some operations, your part editor must retrieve the window pointer from the ODWindow object and use the platform's facilities. In most cases, however, the interface to ODWindow

provides the capability you need for interacting with your windows.

Window State Object

There is a single instantiated window state object (class ODWindowState) for each OpenDoc session. The window state consists of a list of the currently existing window objects. You can access all open windows in a session through the window state.

The document shell and dispatcher use the window state to pass events to parts so that they can activate themselves, handle user input, and adjust their menus as necessary. A part may be displayed in any number of frames, in any window of a document. The dispatcher uses the window state to make sure that it passes events to the correct part, no matter what window encloses the active frame and how many other frames the part has.

Normally, your part editor calls the window state only when it creates new windows, when it needs access to a particular window, and when it needs access to the base menu bar object.

If for some reason your part needs access to all windows, you can create an ODWindowIterator object, which gives you access to all windows referenced by the window state.

Creating and Registering a Window

In order to receive events in a window, you generally must create an OpenDoc window object for it. (You may not be able to create an OpenDoc window for a modal dialog box, but you can pass an event filter routine to it and therefore receive events.) Windows in OpenDoc are created and maintained through the window-state object, which you access through the session object.

You first create a window with platform-specific calls; you then call the window state object to create an OpenDoc window object describing the platform-specific window. You call either of two methods:

- You call the RegisterWindow method of ODWindowState when you create a window that is not a root window or has never yet been written to storage.
- You call the call the RegisterWindowForFrame method of ODWindowState when you create a root window from a previously stored root frame. (RegisterWindowForFrame takes fewer parameters than RegisterWindow because the frame passed in with the method already contains some of the needed information).

A window has an is-root property. If the property is true, the window is a root window, which is the same as a document window. The root part of a root window is the root part of its document, and the document cannot close as long as the root window is open. If a window's is-root property is false, the window may be either a part window that has been opened from a source frame within a root window, or it may be a dialog box, palette, or other utility window. OpenDoc permits multiple document windows for a single document, as long as the root part provides a user interface to support this feature. The document shell closes a document when the document's last document window (root window) is closed.

Windows also have a should-save property that, if true, specifies that the state of the window is saved persistently after the window closes. Usually, only document windows should be marked as should-save.

The creator of a window can specify the view type and presentation of the *root frame* , the frame that displays the root part. The view type specifies whether the root part should draw itself as an icon, and the presentation specifies what kind of appearance the part content should have if not drawn as an icon. View type and presentation are suggestions to the part editor that draws within that frame. View type and presentation are described in more detail in Defining General Display Characteristics.

OpenDoc assumes that each window has a single canvas, which is attached to the window's *root facet* , the facet created for the root frame. On the OpenDoc platform, the root frame in the window has the same shape as the window's content region.

Your part should create windows as invisible and then made visible as described in Opening a Window.

Opening a Window

After creating a window, your part editor typically makes calls to these three methods, in this order:

1. The window's Open method, which creates the root facet
2. The window's Show method, which makes the window visible

3. The window's Select method, which brings the window to the front

---

Window IDs

Your part editor should not maintain references to ODWindow objects for accessing OpenDoc windows, because the document shell or the window state object can close a window and invalidate the reference. Instead, the window state assigns window IDs that are valid for the length of a session. Use the window's GetID method to get the ID of a window when you create it, and then pass that ID to the window state's AcquireWindow method for subsequent access to the window.

Closing a Window

If your part editor needs to close a window programmatically, it calls the window's CloseAndRemove method. That method closes the window, releases the window object and disposes of the platform-specific window structure, deletes the root facet and canvas, and removes the root frame from the document. It also makes any necessary platform-specific calls to dispose of the window itself.

Storing and Retrieving Window Characteristics

Whenever a document is saved, OpenDoc writes certain information into a storage unit of the window's root frame. The window's bounding rectangle, title, and other characteristics are saved in a property of type kODPropWindowProperties in the frame's storage unit.

When you create a root window, you retrieve that information from the stored frame, and use it to specify the platform-specific window's characteristics. You can use functions of the WinUtils utility library (provided with OpenDoc) to extract that information, or you can access the frame's storage unit directly. The kODPropWindowProperties property contains a persistent reference to another storage unit, which contains the following properties:

- kODPropShouldShowlinks
- kODPropSourceFrame
- kODPropWindowHasCloseBox
- kODPropWindowIsFloating
- kODPropWindowIsResizable
- kODPropWindowIsRootWindow
- kODPropWindowIsVisible
- kODPropWindowProcID
- kODPropWindowRect
- kODPropWindowRefCon
- kODPropWindowTitle

Making sure a window is on screen
If your part is a root part that recreates a previously stored document window, you must make sure that the window is visible onscreen. Your document may have been moved from one system to another with a different monitor configuration or size. You may need to move or resize the window to fit its new environment.

Open Method of your Part Editor

Opening your part means creating a window for it and displaying it in the window.

Your part itself initiates the opening of a window when the user selects the **Open as window** command from the View menu (see View Menu), and when it creates its own dialog boxes. Otherwise, your part opens a window only when your part's Open method is called. This is the interface to the Open method:

```
ODID Open(in ODFrame frame);
```

The Open method is called in these circumstances.

- When your part is initially created -when it has no previously stored frame or window information-OpenDoc calls your part's Open method and passes a null value kODNULL for the frame parameter.
- When your part is an embedded part whose frame is selected, and the user chooses the **Open selection** command from the Edit menu, your containing part calls your part's Open method and passes a reference to the selected frame in the frame parameter.
- When your part is the active part, and the user chooses the **Open as window** command from the View menu, you call your part's Open method passing in your frame.
- When your part is the root part of a document being opened, OpenDoc calls your part's Open method and passes a reference to the root frame in the frame parameter.

In your implementation of Open, you can take steps similar to the following, depending on the circumstances under which it was called.

1. If you are creating an initial window (frame = null), skip this step and go to step 2.
    - If you are opening a frame into a part window (frame = an embedded frame), check to see if the window already exists. If you have created the part window previously and saved its window ID, pass that ID to the AcquireWindow method of the window state object. If the method returns a valid window, bring the window to the front and exit. If the window does not yet exist, go to step 2.
    - If you are opening a stored document into a window (frame = a root frame), read in the saved window data from the storage unit of the frame passed to the method (see Storing and Retrieving Window Characteristics).
2. Create a platform-specific window and register it with the window state object, as described in Creating and Registering a Window.
    - If you are opening a stored document into a window, apply the stored characteristics to the platform-specific window. Call the window state object's RegisterWindowForFrame method.
    - If you are opening a frame into a window or creating an initial window, apply your default characteristics to the platform-specific window. Call the window state object's RegisterWindow method.
3. Get the window's window ID and save it for future reference.
4. Open and bring the window to the front, as described in Opening a Window. If you are opening a stored document into a window, skip this step because there may be more than one window to open and OpenDoc determines which window is to be in front.

Note: You are not absolutely required to open a window when your Open method is called. Your part does whatever is appropriate, given its nature and the conditions under which the method is called. For example, if your part is a very simple sound player, it might simply play its sound and never create a window.

Handling Window Events

To receive events in its windows that it creates, your part must create an ODWindow object for each platform-specific window it uses, including dialog boxes (except for modal dialog boxes; see Modal Dialog Boxes).

The document shell handles most window events outside of the content region -for example, events in the title bar or resize box. Nevertheless, the OpenDoc dispatcher first sends a kODEvtWindow event to the root part. If the part editor of the root part wishes to override the action the document shell would otherwise take, the root part can intercept and act on the event.

If your part handles a window event, its HandleEvent method must return true. If it does not handle an event, its HandleEvent method must return false so that the document shell can handle the event.

Resizing

The document shell usually resizes windows, although the root part can intercept and handle the event. The document shell relies on default size limits for windows, so if your part allows-for example-very small window sizes, it may have to intercept this event and handle the resizing itself.

When a window is resized, the active part does not change, but the part editor for the root frame is informed of the resizing through calls by OpenDoc to its FrameShapeChanged and GeometryChanged methods. The root part can then do any necessary invalidation and subsequent redrawing, including creation of new facets if embedded parts have become visible due to the resizing.

Closing

The document shell handles a mouse click in the Close icon of a window or user action of Close or its keyboard equivalent. The document shell closes the window, after which the window cannot be reopened.

If the window is a document window and is the only one open for that document, the document shell closes the document.

If your part editor needs to close a window programmatically, it can call the window's CloseAndRemove method. The window is closed and the window object is released.

Dragging

The document shell handles some platform-specific window-moving actions, such as dragging of a window by its title bar. No event-handling is required of the window's root part.

Parts in other windows may need to be updated because of the window's move; they receive update events as appropriate.

Modal Dialog Boxes

When your part editor displays a modal dialog box, it does not need to create an ODWindow object, as with a regular window. However, it should still request the modal focus (using its own display frame as the modal-focus owner), and it can still receive events by providing an event filter

In addition, to ensure that floating windows are properly deactivated, your part must deactivate the front window before displaying a Modal dialog and it must reactivate the front window after dismissing it.

Your part can create and register its own dialog window, request the modal focus for the window's root frame, and handle the dialog box.

Acquiring and Relinquishing the Modal Focus

A frame displaying a modal dialog box should own the modal focus, a focus type that exists to constrain certain events.

For example, a mouse click outside the frame that has the modal focus still goes to that frame. If your part's frame has the modal focus and the user clicks outside the frame, your part's HandleEvent method is called and passed a facet of kODNULL. The method should check for a null facet in this situation and either alert the user with a beep or dismiss the dialog box, as appropriate.

A click in a frame embedded within the frame that has the modal focus goes to the embedded frame. This behavior facilitates the construction of dialog boxes and other controls from multiple parts.

Your part obtains and relinquishes the modal focus as it does other foci, as described in Requesting Foci and Relinquishing Foci.

In general, your part should not be willing to relinquish the modal focus on request. If your part is displaying a modal dialog, you probably do not want any other modal dialog to be displayed at the same time. To make sure that your part holds on to the modal focus, your part editor's BeginRelinquishFocus method should return kODFalse if the requested focus is kODModalFocus and the proposed new owner of the focus is not one of your own display frames.

When you have finished displaying a modal dialog box, you can directly transfer it to its previous owner by calling the arbitrator's TransferFocus method, as noted in Handling a Simple Modal Dialog Box.

Event Filters

With modal dialog boxes, your part editor's dialog-box event filter controls which events you receive while a dialog box or alert box is being displayed. To pass received null events, update events, and activate events on to OpenDoc or other windows for handling, your event filter can send them to the OpenDoc dispatcher by calling its Dispatch method.

Your event filter should not pass other events, such as mouse events, to the dispatcher.

Handling a Simple Modal Dialog Box

To display a simple modal dialog box or Alert box, you can take these steps:

1. Get a reference to the frame that currently owns the modal focus by calling the arbitrator's GetFocusOwner method. Request the modal focus from the arbitrator, using its RequestFocus method. If you obtain the focus, proceed.
2. Install your dialog event filter function.
3. Create the dialog box, using a Dialog Manager function such as GetNewDialog or a utility function such as ODGetNewDialog (from the DlogUtil utility library provided with OpenDoc). One advantage of using ODGetNewDialog is that it positions the dialog box in relation to your part's document window, rather than to any part windows that may be open.
4. To handle floating windows properly, deactivate the currently active window and any associated floating windows by calling DeactivateFrontWindow method of the window state object.
5. Handle the dialog box with a Dialog Manager call such as ModalDialog. Act on the results and, when finished, dispose of the dialog box with a Dialog Manager call such as DisposeDialog.
6. Reactivate the previously active window (to restore floating windows), by calling the window state's ActivateFrontWindows method.
7. Remove your dialog event filter function.
8. Restore the modal focus to its previous owner by calling the arbitrator's TransferFocus method.

By always saving and restoring the owner of the modal focus, your part can use this approach for nested modal dialog boxes, such as a dialog box that is built from several embedded parts.

Handling a Movable Modal Dialog Box

In OpenDoc, to implement a full-featured movable modal dialog box-that is, one that allows process switching-you must create a window object (ODWindow) to contain it. To display a movable modal dialog box, you can take these steps:

1. Use the appropriate methods to create the structures for the dialog box.
2. Create a window object, using the window state's RegisterWindow method. Give it properties appropriate for your modal dialog, such as floating, nonpersistent and floating.
3. Request the modal focus for the root frame of the dialog window.
4. Adjust menus, as necessary, for the presence of the dialog box.
5. Call the Open, Show, and Select methods of the modal dialog window.
6. Handle events in the dialog box through your normal event-handling mechanism.

To make sure you dismiss the modal dialog box at the right time, you can take actions such as these when you receive a mouse-down event in the dialog box:

1. Determine whether the event applies to your dialog and, if so, what item the user selected.
2. If the user chose to close the dialog box, relinquish the modal focus and call the window's CloseAndRemove method to delete the window and its root frame.
3. Re-enable any menus or menu items that you disabled for display during of the dialog box.

Note: It is also possible to create a modal dialog box that is movable, but does not support process switching. To do do, use a filter function and other functions in a utility library (DlogUtil) provided with OpenDoc.

Modeless Dialog Boxes

Modeless dialog boxes are more like regular windows than modal dialogs are. They can be activated and deactivated, and they need not be dismissed for your part to become active and editable.

Showing the Dialog Box

To display a modeless dialog box in OpenDoc, you must create a window object (ODWindow) to contain it. To display the dialog box, you can take steps such as these:

1. In case the dialog window already exists, try to get a reference to it by passing its ID (previously stored in your part) to the window state's AcquireWindow method. If it does not yet exist, create the platform-specific structures for the dialog box, and create a window object with the window state's RegisterWindow method. Call the window's Open method.

2. Call the window's Show and Select methods to make it visible and active.

3. If you do not already have the window ID of the dialog window, get it by calling the window's GetID method. Save it for use in step 1 the next time the user chooses the action that brings up the modeless dialog box.

Closing the Dialog Box

When the user clicks in the close box of a modeless dialog, you may hide the dialog window, rather than close it, so that it is not destroyed. This is an optimization that allows you to quickly display the dialog box again.

In your part's HandleEvent method, you can respond in this general way to a mouse click within a window's close box:

1. From the frame or facet passed to HandleEvent, obtain information that can identify the window. For example, get a reference to the window object in which the event occurred (by calling the facet's GetWindow method), or examine the frame's presentation or part info data for identifying characteristics.

2. Compare that information to stored information that defines your modeless dialog box. For example, get a reference to your modal dialog's window object (by passing its ID to the window state's AcquireWindow method), for example, or check a stored value that defines your modeless dialog's presentation.

3. If the two are the same, hide the window instead of closing it.

Hiding a Dialog Box when Deactivating a Frame

When your part is deactivated, it should hide any of its modeless dialog boxes.

When your part relinquishes the selection focus, it can get a reference to the dialog window (by passing its ID to the window state's AcquireWindow method), call the window's IsShown method to see if it is currently being shown, and then save that shown state and hide the window.

When your part reacquires the selection focus, it can retrieve a reference to the dialog window by passing its ID to the window state's AcquireWindow method. Then, if the dialog window had been visible at deactivation, your part can once again show it.

Controls

This section discusses what kinds of controls you construct them, and how to handle events within controls It also discusses two specific issues for palettes: how to share them among instances of your part, and how to use them to embed parts within your part.

Design Issues for Controls

Controls in OpenDoc have the same function as in conventional applications: they are graphical objects that allow the user to interact with and manipulate documents in a variety of ways. However, there are some differences:

- In a conventional application, controls typically apply to the entire document. They are always present unless dismissed explicitly. In an OpenDoc document, however, each part may have its own set of controls, and thus controls can appear and disappear rapidly as the user edits. This rapid change can be irritating if not carefully managed.
- In a conventional application, finding the space in which to display a control may be less of a consideration than in OpenDoc. It may be a challenge for small embedded parts to find sufficient space to display rulers, scroll bars, and palettes.
- In OpenDoc, controls can be constructed as independent parts, assemblages of parts, different frames of a single part, or content elements of a part. This can give you more flexibility in constructing user-interface elements than is possible for conventional applications.
- OpenDoc controls can have attached scripts or can communicate with each other, or with other parts, using the OpenDoc extension mechanism. This gives OpenDoc controls the ability to be far more integrated and context-sensitive than standard controls.

Standard types of controls you might wish to include with your parts include:

- Push buttons, radio buttons, and checkboxes
- Scroll bars and sliders, or other gauges
- Pop-up menus
- Rulers
- Tool bars
- Palettes

Rulers usually reflect some settings of the current selection context and contain controls that allow the user to change these settings. Tool bars are like rulers but they often trigger actions instead of changing settings. Status bars display the progress of some long-running operation or suggest actions to users.

In conventional applications, rulers and tool bars commonly occupy the margins of the window. In an OpenDoc document, controls can be displayed at the margins of the window, in separate frames outside the embedded frame they apply to, or within the frame they apply to.

A ruler for a text part, for example, can be an additional frame associated with the part. Events in the ruler are handled by the part's editor. Alternatively, the ruler may be its own part with its own editor. In this case, the text part editor must maintain a reference to the ruler part and be able to communicate with it through semantic events or some other extension mechanism.

A ruler that is a separate part can have its own embedded parts, such as buttons. The ruler part must then be able to communicate with its embedded controls as well as with the part that it services.

Palettes often contain editing tools but can also contain choices for object attributes like color or line width. Palettes commonly float freely beside or over the document, although they can also be fixed at the window margins. Palettes might also pop up, pull down, or be torn off of menus.

If all parts in a document can communicate with each other, they can coordinate the drawing and hiding of palettes or other controls to avoid irritating the user. For example, all parts in a document can share a single palette, within which the various parts negotiate for space and in which they draw only those tools that must change as the active part changes.

Handling Events in Controls

How you handle events in a control are handled depends on how you design the control:

- If a control like a ruler is within the active frame, it might not have its own frame, and the active part editor handles any events in the ruler.
- If a control has its own frame, it might be another display frame of the active part. In this case also, the active part editor handles the event in the control.

- If a control is a separate part, its own part editor handles the event and updates the state of the part, which might trigger scripts or calls to other parts' extension interfaces. Likewise, the part itself can receive queries from other part editors in the form of semantic events or calls to its extension interface.

Sharing Palettes and Utility Windows

The user-interface guidelines state that you should hide any visible palettes, modeless dialog boxes, or other utility windows when your part becomes inactive or its document closes, and restore them-in the same positions-when the part becomes active once again or the document reopens. In addition, if your part editor maintains multiple parts in a document and the active state switches from one to another of them, any visible utility windows that apply to both parts should remain steadily visible, without any flicker caused by hiding and immediate restoring.

One way to implement this behavior for a utility window is to follow, in general, steps such as these:

- Make the window globally accessible to your parts by keeping its reference and its state in an object that each of your parts acquires on initialization and releases on closing.
- Have the global object create the window in a normal manner the first time it is to be displayed. When the user closes the window, the global object can simply hide it by calling its Hide method. If the user subsequently needs to display the window again, the global object can show it again by calling its Show method.
- When your part relinquishes the selection focus, its CommitRelinquishFocus method can check to see if the part that is receiving the focus also belongs to your part editor, and has a presentation that uses the same utility window. There are a number of ways to check this, such as by examining the presentation, part info, or even the part kind associated with the newly active frame.
- When your part acquires the selection focus, it notifies the global object of that fact. The global object, in turn calls the ChangePart and SetSourceFrame methods of the utility window's frame to assign the new part and frame to the utility window. The new part can then adjust the content of the utility window if needed, and also show or hide other palettes or dialogs.

Using a Tool Palette to Embed Parts

An example of an OpenDoc-specific use for controls is to allow the user to embed parts by selecting items from a palette. Using a palette in this way, your part can create embedded parts from scratch, rather than through reading in existing part data.

Your part can provide a palette, menu, or dialog box from which the user selects an item that specifies a part kind. As in many conventional applications, your palette could display a set of tools to the user-drawing tools, painting tools, text tools, and so on. In this case, however, selecting an item from the palette actually means that a new part of that kind is to be embedded in your part.

The items could represent existing template documents in the user's system, or they could simply represent individual part kinds for which editors exist. If you are creating parts from scratch, follow steps such as these once the user has made a selection from your palette:

- Create the new part by calling your draft's CreatePart method, passing it the part kind that the user selected.
- Call the new part's Externalize method (see The Externalize Method), so the part can create and write initial data to the properties in its storage unit.
- Create a new embedded frame for the part, as described in Creating a New Embedded Frame.
- Give the new frame the proper link status, as described in Frame Link Status.
- If the new frame is visible, assign facets to it, as described in Adding a Facet.
- Notify your containing part and your draft that there has been a change to your part's content; see Making Content Changes Known.

Menus

At any given moment while an OpenDoc document is open, responsibility for the menu bar is shared among three entities. The operating

system provides any system-wide menus, the OpenDoc document shell creates the Document menu, Edit menu, View menu and Help menu. Individual part editors can create other menus as needed. (Part editors can also, with restrictions, add appropriate items to the Document and Edit menus).

Different platforms have different conventions for enabling and disabling menus and menu items. In an OpenDoc document, the document shell, the root part, and the part with the menu focus together control which menu commands are available to the user. As each part becomes active, the OpenDoc document shell and the root part update their own menu items, and the active part editor takes care of the rest.

Basic event handling for menu events is described in Menu Events. When the user chooses a menu item, the document shell either handles the command itself or dispatches a menu event to the active part; the part receives the event as a call to its HandleEvent method.

This section discusses general issues of setting up and working with menus and then describes how to handle individual menu events for the standard OpenDoc menus (the Document menu and the Edit menu):

- Document
- Edit
- View
- Help

Setting Up Menus

This section describes how your part editor can set up and use menus and menu items.

Base Menu Bar

When it first opens a document, the document shell creates a menu bar object (type ODMenuBar) and installs it as the base menu bar by calling the window state's SetBaseMenuBar method. The base menu bar contains different menu and items on different platforms, but the Document menu and the Edit menu are always installed.

Base Pop-Up Menu

On the OS/2 platform, the document shell also creates a pop-up menu object (type ODPopup) and installs it as the base pop-up menu by calling the window state's SetBasePopup method. The base pop-up menu always installs the **Open as properties**, **Show as**, and Help menu items.

Adding Part Menus to the Base Menu Bar

When your part initializes itself, or when it first obtains the menu focus, it should create its own menu bar object by:

1. Copying the base menu bar using the window state's CopyBaseMenuBar method.
2. Adding its own menu structures, by using menu-bar methods such as AddMenuBefore and AddMenuLast.

If absolutely necessary, your part editor can add items to the end of the Document and Edit menus, but you should not alter the existing items.

Menu IDs

You must identify each menu with a menu ID. A menu ID is a positive short; negative values are reserved by the operating system.

All menus in the menu bar must have unique menu IDs. Therefore the document shell, the active part, and any shell plugs-in or services that also have menus must cooperate to ensure that there are no conflicts. Please follow the conventions listed in the following table list to ensure that your menu IDs do not conflict with those of others.

Type of Software	Menu ID Range
Document shell	0x0000 – 0x2FFF
Part editors (when root)	0x3000 – 0x3FFF
Shell plug-in/services	0x4000 – 0x4FFF Note: These menu IDs must be dynamically assigned.
Part editors	0x5000 – 0x7FFF

A shell plug-in or service may have to adjust its menu ID dynamically at runtime, because another service or plug-in with that menu ID may already be installed. The plug-in should choose an ID, look for an installed menu with that ID, and-if it is found-add 1 to the ID and try again.

Obtaining the Menu Focus

When your part activates itself, it should request the menu focus (along with other foci) if it wants to use menus. See Requesting Foci for more information.

Enabling and Disabling Menus and Commands

When the user clicks in the menu bar, the OpenDoc dispatcher determines which part has the menu focus and calls that part's AdjustMenus method. It also calls the root part's AdjustMenus method, if the root part does not have the menu focus.

Your part's AdjustMenus method can use methods of the menu bar object such as EnableCommand or EnableAndCheckCommand to change the appearance of your menu items, or it can make platform-specific calls to directly enable, disable, mark, or change the text of its menu items.

Your AdjustMenus method typically acquires the clipboard focus and enables the **Cut**, **Copy**, **Paste**, and **Paste as...** items in the Edit menu.

Menus and Read-Only Documents

When your part permissions (see Drafts) specify that your document is read-only, your part editor needs to disable these menu commands:

- **Cut**, **Paste**, **Paste as...**, and **Delete** items in the Edit menu
- any part-specific content-editing commands

This situation can occur when the user views an early draft of a document, or when a document is stored on read-only media.

Part viewers should disable these items and commands at all times.

Menus and the Root Part

In all OpenDoc documents, the root part is responsible for printing. The root part therefore should handle the **Print document** item from the

Document menu, even if an embedded part has the menu focus. If the root part is a container, it is also responsible for opening an icon, tree, or details view of itself.

To allow the root part access to these menu events, the dispatcher passes menu events to the root part if they are not handled by the part with the menu focus. Also, OpenDoc calls the root part's AdjustMenus method before it calls the AdjustMenus method of the part with the menu focus, so that the root part can adjust the state of those menu items.

When your part's AdjustMenus method is called, it should check to see if your part is the root part and whether it has the menu focus. If it is the root part but does not have the menu focus, it should adjust only the **Open as** and **Print document** items from the Document menu. If it is an embedded part with the menu focus, it should not adjust those items.

Likewise, when your part's HandleEvent method is called, it should first check to see if your part is the root part. If it is the root part, HandleEvent should return false if it is passed any menu events other than the **Print document** item and the **Icons**, **Tree**, or **Details** items of the **Open as** submenu in the Document menu. If it is an embedded part, HandleEvent should return false if it is passed **Page setup** or **Print document**.

Document Menu

This section describes how your part editor should interact with the Document menu. The OpenDoc document shell handles most Document menu commands, as described in The Document Shell and the Document Menu. Individual part editors must respond only to the **Open as** and **Print document** commands.

Open as

The user chooses the **Open as** (in an OpenDoc document) into its window.

The root part handles this command. It should display one of the user selected views (icons, tree, or details) using the ODViewExtension class.

Your part does not need to support this command if it is not a container.

Print document

The root part of the window handles the **Print document** command. Root parts are responsible for defining the printing behavior of their documents. When the user chooses this command and your part is the root part of the active window, OpenDoc passes the command OD_PRINT to your part's HandleEvent method. If the root part does not handle the event, the document shell prints the document. The document shell prints only the visible content of the root frame and its embeds. If your part supports multiple pages or its content area is greater that its frame, you need to handle the print command yourself.

Your routine to handle the **Print document** command should set up for printing.

Edit Menu

Most items in the Edit menu are handled by individual part editors. Most apply to the current selection in the currently active part.

Choices on the Edit menu are as follows:

- Undo
- Redo
- Create
- Cut

- Copy
- Paste
- Paste as...
- Paste link
- Break link
- Delete
- Select all
- Deselect all
- Open selection
- Selection properties
- Show selection as

The following figure shows the Edit menu.

Each item is explained in detail in the following sections.

Undo

The user selects **Undo** from the Edit menu in order to reverse recently enacted user actions, including previous uses of **Undo** or **Redo**. **Undo** reverses the effects of the last undoable user action and restores all parts to their states before that action. Not all user actions are undoable. OpenDoc supports multiple levels of **Undo**. For example, if the user selects **Undo** three times in succession, then the last three undoable actions are undone in order.

The document shell handles the **Undo** item, passing control to the Undo object. The Undo object, in turn, calls the UndoAction method of any part editors involved in the **Undo**.

Your part should respond to Undo as described in Undo.

Redo

The user selects **Redo** from the Edit menu in order to reverse the last use of **Undo**. **Redo** is available only if the last undoable user action was **Undo** or **Redo**. **Redo** reverses the effects of the last undoable user action, restoring all parts to their states before the **Undo** action. OpenDoc supports multiple levels of **Redo**. Each time the user selects **Redo**, the previous **Undo** is redone. This works only if the user had invoked **Undo** multiple times in succession.

The document shell handles the **Redo** item, passing control to the Redo object. The Redo object, in turn, calls the RedoAction method of any part editors involved in the **Redo**.

Your part should respond to **Redo** as described in Undo.

Create

The user selects **Create** to create a new part of the same type as the selected part. The part will be created on the clipboard. When **Create** is selected, the **Paste link** menu item and the **Paste Link** push button in the Paste As... dialog should be grayed because linking is not possible at this point.

Cut, Copy, and Paste

The user selects **Cut**, **Copy**, or **Paste** from the Edit menu in order to place data onto the clipboard or retrieve data from the clipboard.

The **Cut**, **Copy**, and **Paste** choices are familiar mechanisms for copying and moving content around in the system. In OpenDoc, function has been added for handling embedded parts. The function added consists of:

- Operating on parts in addition to intrinsic content. The parts may be represented as frames or icons.
- Making embed versus incorporate decisions.
- Wrapping intrinsic content inside a part in certain circumstances.

Your part should disable the **Cut** and **Paste** items when its draft permissions are read-only; see Menus and Read-Only Documents.

Paste as...

The user selects **Paste as...** from the Edit menu in order to specify the manner in which clipboard data is to be pasted into the active part.

The **Paste as...** choice displays a dialog box that allows the user to specify the data format for pasting the clipboard into the destination. The clipboard content is converted to the user-specified format, and links to it can be created. Edit Menu lists the kinds of pasting that the user can specify. When the user selects the command, OpenDoc passes the command information to your part's HandleEvent method.

Your routine to handle the **Paste as...** command should prepare to read from the clipboard, like this:

1. Acquire the clipboard focus and gain access to its content storage unit, following the initial steps described in Pasting from the Clipboard.
2. Display the Paste As... dialog box, by calling the ShowPasteAsDialog method of the clipboard object. Pass the function the active frame into which the paste is to occur.
3. If the method returns kODTrue, the user has pressed the **OK** button; use the results of the interaction to determine which kind of pasting action to take. Then read the appropriate kind of data from the clipboard, continuing with the procedures shown in Pasting from the Clipboard.

Paste link

The user chooses the **Paste link** command from the Edit menu in order to insert the link from the clipboard into your part.

In your routine to handle paste link, you should:

1. Create an empty "linkspec"
2. Read in the "linkspec" from the clipboard
3. Create a link from the "linkspec"

Break link

The user selects **Break link** to copy the source information to the document and break the link with the source. The action cannot be undone.

A confirmation dialog will be given before the link is broken. The **Break link** choice is for destination links only.

Delete selection

The user selects **Delete selection** to delete the selected content from the active part.

Your routine to handle the **Delete selection** command should remove the items that make up the selection from your part content. That may involve deleting embedded parts as well as intrinsic content.

Select all

The user selects **Select all** from the Edit menu to make the current selection encompass all of the content of the active part.

Your routine to handle the **Select all** command must include all of your part's content in the selection structure that you maintain and it must highlight the visible parts of it appropriately.

Deselect all

The user selects **Deselect all** from the Edit menu to deselect all the content of the active part.

Your routine to handle the **Deselect all** command must include all of your part's content in the selection structure that you maintain and it must highlight the visible parts of it appropriately.

Selection properties

The user chooses the **Selection properties** command from the Edit menu to display the Properties notebook containing standard information about the current selected embedded parts. When the user chooses the command, OpenDoc passes the command information to your active part's HandleEvent method. Your routine to handle the **Selection properties** command should display the Properties notebook that describes the characteristics of the current selection.

When handling the **Selection properties** command, if the current selection is an embedded frame border, display the Properties notebook for the part in that frame, using the ShowPartFrameInfo method of the info object (class ODInfo). You obtain a reference to the Info object by calling the session object's GetInfo method. OpenDoc handles all changes made by the user and updates the information in the storage units for the embedded part and frame, including performing any requested translation.

OpenDoc determines the information it displays in the Properties notebook by reading the part's Info properties, the set of properties-separate from the part's contents-that can be displayed to, and in some cases, changed by the user. All stored parts include them. If your part creates an extension to the Properties notebook (see The Settings Extension), it can define and display additional properties. Your routine should support both single and multiple selection.

Open selection

The user chooses the **Open selection** command from the Edit menu to open the selected embedded parts within a window. When the user chooses this command, OpenDoc passes the command to your active part's HandleEvent method. Your routine to handle **Open selection** should do the following:

1. From your private data structures, determine which of your embedded frames is the selected one. (Each embedded part, even if displayed in an icon view type, has a frame).

2. Call the AcquirePart method of the selected frame, followed by the Open method of the part returned by the AcquirePart method.

The Open method is described in Open Method of your Part Editor.

Your routine should support both single and multiple selection.

Show selection as

The user chooses the **Show selection as** command from the Edit menu to change the view type of the selected embedded parts into large icons, small icons, thumbnails, or frames. When the user chooses this command, OpenDoc passes the command to your active part's HandleEvent method. Your routine to handle **Show selection as** should do the following:

1. For each embeded selection frame, obtain a reference to the part that owns the frame.
2. Call that frame's ChangeViewType method, passing in the view type choosen by the user.

View Menu

The View menu shows the views of the active part or controls the display of related windows like tool bars or palettes. All container parts should support Icons, Tree, and Details views.

The View choices are as follows:

- Show frame outline
- Open as
- Properties
- Show as
- Show links

The following figure shows the View menu.

Each item is explained in detail in the following sections.

Show frame outline

The user selects this command to reposition your part's content-that is, to change the portion of it displayed in the frame in the document window. In the part window, display a 1-pixel wide black-and-white border around the content currently visible in the display frame in the document window, and allow the user to drag the frame outline.

This appears in the View menu for parts opened into a window. When it is selected, an outline of the frame is shown to indicate the portion of a part's contents that is displayed in its frame. The user can drag this outline to adjust the visible region of the part. **Show frame outline** toggles with **Hide frame outline**.

Your part should add the **Show frame outline** command to the View menu when your part's content area is greater than the area of its display frame, the frame is opened into a part window, and the part window is active.

Open as

The user chooses the **Open as** command from the View menu to display an icon, tree or details view of your part and its embeds. If your part is a containing part, you should support these views.

Display the view for your frame by calling the DisplayView method of the ODViewExtension class, passing in the view type selected by the user.

Properties

The user chooses the **Properties** command from the View menu to display a Properties notebook for your part.

The Properties notebook is displayed for your frame using the ShowPartFrameInfo method of the ODInfo class. You obtain a reference to the info object by calling the session object's GetInfo method. OpenDoc handles all changes made by the user.

Show as

The user chooses the **Show as** item from the View menu to change the view type of your part's frame. Your part should call its SetViewType method, passing in the view type selected by the user.

Show links

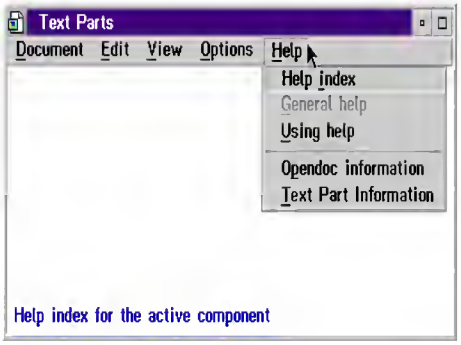
The user selects this command to show the link border for all links in the active part. The **Show links** menu item toggles with **Hide links**.

Help Menu

The Help menu choices are as follows:

- Help index
- General help
- <part> information

The following figure shows the Help menu.



Help index

Index for the active part. When the part receives this command, it should display its index by calling the DisplayHelpIndex method of the ODHelp class. A part obtains a reference to the ODHelp class by calling the GetHelp method of the ODSession class.

General help

A general description of the part. When a part receives this command, it should display its general help by calling the DisplayHelp method of the ODHelp class. A part obtains a reference to the ODHelp class by calling the GetHelp method of the ODSession class.

---

information

This menu provides information about the active part. This menu is optional and should be implemented by the part.

Pop-Up Menu

The pop-up menu choices are as follows:

- Open as
- Properties
- Show as
- Help

Additional actions that a part can perform on the part should also be listed in the pop-up menu. Related choices should be grouped together. The commands generated from the pop-up menus are the same as those generated from the menu bar.

- View choices (for example, **Open as**, **Properties**, and so forth)
- Clipboard/data transfer (for example, **Cut**, **Paste**, **Delete**, and so forth)
- Printing
- Part actions
- Container actions (for example, **Align**)
- Help

Pop-Up Menus and Multiple Selection

In the Workplace Shell on OS/2 when you select multiple objects and then bring up a pop-up menu, you get the intersection of choices you can perform. In the first release of OpenDoc, limited support for this feature will be provided. There are three different cases to consider:

- When the multiple selection includes intrinsic content only
- When multiple embedded parts are selected only
- When both intrinsic content and embedded parts are selected

In the first case, the part editor should be able to provide intersection of choices that apply to all the intrinsic data that is selected.

Part developers are free to add more choices to the pop-up menu. For example, if the parts are members of a suite of parts, and the part editor can correctly determine the intersection of choices, then they should do so.

In the third case, at a minimum, any container choices (for example, Align) that can be performed on parts should be displayed. Part editors can add more choices as long as they apply to both the part and the selection.

Accelerator Table Sharing

OpenDoc creates a default accelerator table containing pre-defined accelerator keys for OpenDoc. Parts can add to the accelerator table by following these steps:

1. Fill in an ODACCEL structure with the accelerator information.
2. Call the AddToAccelTable method of the ODMenuBar class.

The AddToAccelTable method can add up to 20 accelerator keys to the accelerator table with each method call.

Status Line

There are two uses for the status line. One is to display help for menu items. The other is to display information or progress to the user for actions occuring on within the active part.

To use the status line as help for menu items, a part should call the SetMenuItemStatusText method of the ODMenuBar class, passing in the menu ID and the text to be displayed. OpenDoc displays the text when the menu item is selected by the user.

To use the status line to display information to the user, a part needs to do the following:

1. Call the session's AcquireExtension method for the status line extension.
2. Call the StatusLineExtension method of the ODArbitrator class.
3. Display the text using the status line extension's SetStatusLineText method.
4. Release the focus when done.

Undo

The **Undo** command (and its reverse, **Redo**) is a common feature on many software platforms. It is designed to allow users to recover from errors that they recognize soon after making them. OpenDoc offers a flexible and powerful **Undo** capability.

Multilevel, Cross-Document Capability

Most systems support only a single-level of **Undo**; that is, only the most recently executed command can be reversed. Therefore, in most platforms, **Undo** is restricted to a single domain. A complex operation that involves transferring data from one document to another, for example, cannot be completely undone.

OpenDoc, by contrast, supports multiple levels of **Undo** and **Redo**.; there is no limit, other than memory constraints on the user's system, to

the number of times in succession that a user can invoke the **Undo** command. OpenDoc also allows for inter-document **Undo** and **Redo**. Together, these enhancements give users greater flexibility in recovering from errors than is possible with simpler **Undo** capabilities.

The OpenDoc **Undo** feature does not offer infinite recoverability. Some user actions clear out the **Undo** action history, the cumulative set of reversible actions available at any one time, resetting it to empty. A typical example is saving a document; the user cannot the document by choosing **Undo**, and no actions executed prior to its saving can be undone. As a part developer, you decide which of your own actions are undoable, which ones are ignored for **Undo** purposes, and which ones reset the action history. In general, actions that do not change the content of a part (such as scrolling, making selections, and opening and closing windows) are ignorable and do not need to be undoable. Non-undoable actions that require clearing the action history are few; closing your part (see Closing your Part) is one of the few.

When the user selects **Redo**, the effects of the last **Undo** are reversed. **Redo** is available only if the user has previously chosen **Undo**, and if nothing but ignorable actions have occurred since the user last chose **Undo** or **Redo**. Like **Undo**, **Redo** can be chosen several times in succession, limited only by the number of times **Undo** has been chosen in succession. As soon as the user performs an undoable action (such as an edit) OpenDoc clears the **Redo** history and **Redo** is no longer available until the user chooses **Undo** once again.

To implement this multilevel **Undo** that spans different parts and different documents, OpenDoc maintains a centralized repository of undoable actions, stored by individual part editors as they perform such actions. **Undo** history is stored in the Undo object, an instantiation of the class ODUndo, created by OpenDoc and accessed through the session object's GetUndo method. Your part editor needs access to the Undo object to store undoable actions in it or retrieve them from it.

Importance of Multilevel Undo
Your part must support multiple levels of **Undo**. If your part supports only a single-level **Undo** command, other parts that support multilevel undo will lose their **Undo** history when they interact with your part. You should support multiple levels of **Undo** if you support **Undo** at all.

Implementing Undo

To implement support for **Undo** in your part editor, you need to be able to save information to the Undo object that allows you to recover previous states of your part, and your part editor needs to implement the following methods:

- UndoAction
- RedoAction
- ReadActionState
- WriteActionState
- DisposeActionState

The Undo object calls your part's UndoAction and RedoAction methods when the user chooses, respectively, the **Undo** and **Redo** items from the Edit menu.

The Undo object calls your part's DisposeActionState method when an **Undo** action of yours is removed from the **Undo** action history. At that point, you can dispose of any storage needed to perform the specified **Undo** action.

WriteActionState and ReadActionState
The WriteActionState and ReadActionState methods of ODPart exist to support a future cross-session **Undo** capability. OpenDoc can call these methods when it needs you to store persistently or retrieve your undo-related information. Version 1.0 of OpenDoc does not support cross-session **Undo**, however, so you need not override these methods.

Adding an Action to the Undo History

If your part performs an undoable action, it should call the Undo object's AddActionToHistory method. Your part passes an item of action data to the method; the action data contains enough information to allow your part to revert to the state it occupied just prior to the undoable action. The action data can be in any format; OpenDoc adds it to the action history in the Undo object and passes it back to your part if the user asks your part to perform an **Undo**.

The item of data that you pass to AddActionToHistory must be of type ODActionData, which is a byte array. When you create the item, you can either copy the data itself into the byte array or you can copy a pointer to the data into the byte array; either way, you then pass the byte array to AddActionToHistory.

You also pass two user-visible strings to the AddActionToHistory method. The strings have the text that you want to appear on the Edit menu, such as "Undo cut" and "Redo cut". You must also specify the data's action type, which is described under Creating an Action Subhistory.

In general, you add most editing actions to the action history, unless their data is so large that you cannot practically recover the pre-action state. Opening or closing a document, or performing ignorable actions such as scrolling or selecting, should not cause you to add an action to the action history.

You decide what constitutes a single action. Entering of an individual text character is not usually considered an action, although deleting or replacing a selection usually is. The sum of actions performed between repositionings of the insertion point is usually considered a single

undoable action.

Adding Multistage Actions

Some transactions, such as drag-and-drop, have two stages-a beginning and an end. To add such a transaction to the **Undo** history, your part must define which stage you are adding, by using the action types kODBeginAction and kODEndAction. (For undoable actions that do not have separate stages, you specify an action type of kODSingleAction when calling AddActionToHistory).

In the case of drag and drop, the sequence is like this:

- The source part calls AddActionToHistory at the beginning of a drag, specifying an action type of kODBeginAction to add a **beginning action**.
- The destination part calls AddActionToHistory when the drop occurs, adding a single-stage action to the undo history by specifying kODSingleAction.
- The source part once again calls AddActionToHistory after the completion of the drop (when the drag-and-drop object's StartDrag method returns), specifying an action type of kODEndAction to add an **ending action**.

Similarly, if the user selects the Paste with link checkbox in the Paste As dialog box (see the figure in Handling the Paste As Dialog Box), the part receiving the paste and creating the link destination adds the begin action and end action, whereas the source part adds a single action when it creates the link source.

As the case of drag and drop demonstrates, parts can add single-stage actions to the undo history during the time that a two-stage undoable action is in progress that is, between the times AddActionToHistory is called with kODBeginAction and with kODEndAction, respectively. These actions become part of the overall undoable action.

Menu strings and two-stage actions
The strings you can provide when calling AddActionToHistory are ignored if the action type is kODBeginAction. Furthermore, any strings you provide for subsequent single actions are ignored. Once you have created a beginning action, only the strings passed for the action type kODEndAction appear in the menu.

You can remove an incomplete two-stage action from the undo action history without having to clear the entire history by using the AbortCurrentTransaction method. See Clearing the Action History.

Creating an Action Subhistory

In general, the undoable actions a user performs while a modal dialog box is displayed-as long as the actions do not clear the **Undo** action history of your document-should not affect the action history previous to that modal state. After dismissing the dialog box, the user should be able to undo actions performed before the modal dialog box was displayed-even though at this point the user could not undo actions taken while the modal dialog box was open.

To implement this behavior, you can put a mark into the action history that signifies the beginning of a new action subhistory. To do so, call the MarkActionHistory method of the Undo object. To clear the actions within a subhistory, you specify kODRespectMarks when you call the ClearActionHistory method. To clear the whole action history, specify kODDontRespectMarks instead.

For every time you put a mark into the action history, you must make sure there is an equivalent call to ClearActionHistory to clear that subhistory.

Undoing an Action

When the user chooses to **Undo** an action added to the action history by your part, OpenDoc calls your part's UndoAction method, passing it the action data you passed in earlier. Your part should perform any reverse editing necessary to restore itself to the pre-action state. (When a two-stage transaction involving your part is undone, OpenDoc calls both your part and the other part involved, so that the entire compound action is reversed).

Redoing an Action

When the user chooses to **Redo** an action of your part, OpenDoc calls your part's RedoAction method, passing it the same action data that you had passed in earlier when the original action was performed. In this case, your task is to re-achieve the state represented by the action data, not reverse it. (When a two-stage transaction involving your part is redone, OpenDoc calls both your part and the other part involved, so that the entire compound action is reversed).

Clearing the Action History

As soon as your part performs an action that you decide cannot be undone, it must clear the action history by calling the Undo object's ClearActionHistory method. Actions that cannot be undone, and for which the action history should be cleared, include saving a changed document and performing an editing operation that is too large to be practically saved. Undoable actions such as normal editing and performing ignorable actions, such as scrolling or selecting, should not affect the action history.

If your part initiates a two-stage action and, because of an error, must terminate it before adding the second stage to the action history, it can remove the incomplete undo action by calling the undo object's AbortCurrentTransaction method. AbortCurrentTransaction clears the beginning action and any subsequent single actions form the undo stack, without clearing the entire action history.

OpenDoc itself clears the action history when it closes a document.

Undo and Embedded Frames

Every frame has a flag, the in-limbo flag, that determines whether the frame (and any of its embedded frames) is actually part of the content of its draft, or whether it is currently "in limbo" (referenced only in undo action data). When a frame is initially created, whether through CreateFrame or by cloning, its in-limbo flag is cleared (kODFalse).

Whenever your part cuts, pastes, drags, or drops an embedded frame, you must keep a reference to that frame in an undo action. When you first perform the data transfer involving the frame, and when you perform subsequent actions with the frame (including undo and redo), you must set the frame's in-limbo flag accordingly. OpenDoc requires the correct flag settings to properly handle moved and deleted objects.

The following table shows how to set the flag properly for each situation, and how to dispose of the frame referenced in your undo action when you part's DisposeActionState method is called.

Action	Set flag to...	If Undone, set flag to...	If Redone, set flag to...	When DisposeActionState is called
Creating a frame	(leave as is)	kODTrue	kODFalse	If kODTrue, call Remove; If kODFalse, call Release
Deleting, cutting, or starting a drag-move*	kODTrue	kODFalse	kODTrue	If kODTrue, call Remove; If kODFalse, call Release
Copying a frame	(leave as is)	(Note1)	(Note1)	(Note1)
Pasting or dropping a moved (not copied) frame	kODFalse (but save prior value)	(restore prior value)	kODFalse (but save prior value)	If kODFalse, call Release; If prior value = kODTrue, call Release; Otherwise, call Remove
When StartDrag returns	kODFalse (if drag was a copy or if drop failed; else leave as is)	(leave as is)	kODFalse (if drag was a copy or if drop failed; else leave as is)	(nothing)

Note1: Do not create an undo action for this situation.

* "Starting a drag-move" in this case means starting a drag in any situation in which a move is a possible outcome.

You set the value of a frame's in-limbo flag by calling its SetInLimbo method and passing either kODTrue or kODFalse. You can determine the value of a frame's in-limbo flag by calling its IsInLimbo method.

When you re-embed a frame in performing an undo or redo action. You must also reset its link status appropriately. See Frame Link Status for more information.

Storage

This is the fifth of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes the external storage facilities OpenDoc uses and provides. OpenDoc relies upon the native file-storage facilities of each of the individual OpenDoc platforms. The same fundamental storage concepts and structures are used consistently for document storage and for data transfer (clipboard transfer, drag-and-drop, and linking, described in the next chapter).

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. Additional concepts related to your part editor's run-time environment are found in OpenDoc Run-Time Features.

This chapter introduces the general architecture of the OpenDoc storage system, including how parts are stored, and then describes how your OpenDoc part editor can use that architecture to store and retrieve its own content.

OpenDoc Storage System

The OpenDoc storage system is a high-level mechanism for persistent or ephemeral storage that enables multiple part editors to effectively share a single document. The storage system is implemented on top of the native storage facilities of each platform that supports OpenDoc.

The OpenDoc storage system effectively gives each part its own data stream for storage and supports reliable references from one stream to another. The system includes a robust annotation mechanism that allows many pieces of code to access information about a given part without disturbing its format.

The identity of a part is consistent within a session; it is unique within its draft and testable for equality with other part identities. Parts can persistently reference other objects in the same draft, including other parts, thus allowing run-time object structures to be saved and reconstructed in a future session.

Parts have more than just their content to store persistently. *Properties* are used to store both the content of the part and supplemental information. OpenDoc defines a standard set of properties for all parts, and you can assign additional properties to a given part. The name of the preferred part editor is one example of a standard property that can be stored with a part.

The storage system allows OpenDoc to swap parts out to external storage if memory is required for other parts. When a part is first needed, its part editor reads it from external storage into memory. When it is no longer needed, the draft releases it. If the part is needed again at a later time, the storage system can either return the in-memory part to the part editor or-if the part has been deleted because of low memory-bring the part once again into memory from storage. When it reads or writes its parts, your part editor may not know whether the data is currently being transferred to or from memory, or to or from external physical storage.

Storage Units, Properties, and Values

The OpenDoc storage system is not an object-oriented database. It is a system of structured storage, in which each unit of storage can contain many streams. This design eases the transition for developers working with existing code bases, which generally assume stream-based I/O.

The controlling storage object is the *storage system* object, which instantiates and maintains a list of containers. The containers are objects in the *container suite* , a platform-specific storage implementation. Version 1.0 of OpenDoc is released with the *Bento container suite* , a container suite that can be used with or independently of OpenDoc.

Each container object in a container suite can hold one or more document objects, each of which, in turn, contains one or more draft objects. Each draft contains a number of storage unit objects, each of which is much like a directory structure in a typical file system (although multiple storage units might be physically stored in a single file, or perhaps not stored in files at all). Storage units hold the streams of stored data.

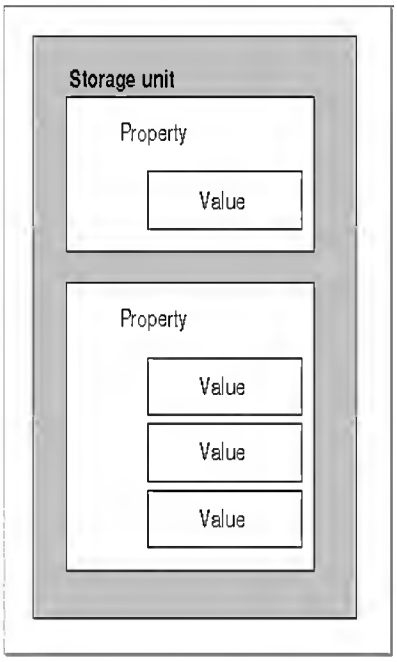
The run-time relationships of these storage objects are diagrammed in the following figure. the figure in section Document Storage

---

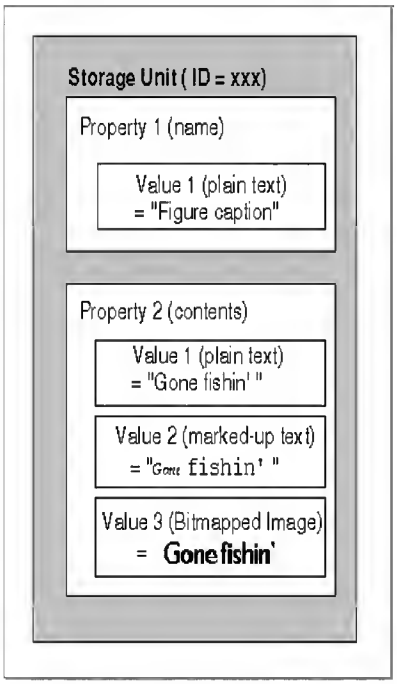
Storage-Unit Organization

You can visualize a storage unit as a set of properties, or categorized groups of data streams. A property is identified by its *property name* , an ISO string. Each property consists of a number of streams called *values* , each of which stores a different representation of the same data, in a format identified by a named type (a *value type* , also an ISO string). Thus there can be several properties (named categories) in a single storage unit, and several values (typed streams) in a single property.

The following figure shows these relationships through a simplified diagram of the organization of a storage unit.



The following figure is a simplified diagram of a specific storage unit in use. The storage unit contains a figure caption. The part that owns this storage unit has created a contents property, which contains the primary data of this storage unit, as well as a name property, which names the item stored in this storage unit. This particular storage unit has three representations of the contents property (the text of the caption) as three different types of values: its own custom styled text, plain text in a standard (international) text format, and a bitmap. The storage unit has only one representation of the name property.



A caller that accesses this storage unit can read or write just the data of immediate interest. In the storage unit of the previous figure, for

example, a caller could access only the value representing one of the data formats in the contents property without having to read the rest of the storage unit into memory. A caller could learn the part kinds of the data stored in the storage unit without having to read any of the contents into memory, even if the part editor that stored the data is not present.

Fundamental to the OpenDoc storage system is the concept that all values within a property are analogous, or equivalent, representations of the same data. Every value in the contents property in the previous figure, for example, is a complete representation of the item's contents, (the caption text), expressed in the type of data that the value holds.

Values in a storage unit can include references to other storage units, as described in Persistent References. Thus, storage units can be arranged in ordered structures such as hierarchies; OpenDoc stores the embedding structure of a document in this way.

Standard Properties

A basic feature of the OpenDoc storage system is that storage units are *inspectable* ; that is, one can extract the data (values) of the various properties of a storage unit without first having to read the entire file or document that the storage unit is part of, and without having to understand the internal format of the data in any of the values. For example, if the storage unit for an embedded part includes a property that holds the part's name, you can extract that name without having to read in all the data of the part, even if you do not understand how to read or display the part's contents.

To make the inspectability of storage useful across parts, documents, and platforms, OpenDoc publicly specifies certain standard properties, with constant definitions such as kODPropContents and kODPropName, that all part editors can recognize. Documents, windows, frames, and parts all have basic structures, defined by properties, that are accessible from both the stored and in-memory states. This accessibility allows OpenDoc and part editors to read only the information that is needed in a given situation and thus increase performance. The table in What a Draft Contains and the table in Info Properties list many of the common standard properties that might be found in storage units.

Using storage-unit methods, you can access the values of any known property of any storage unit.

OpenDoc ISO String Prefix

All public ISO string contants defined by OpenDoc include a prefix that identifies them as OpenDoc ISO strings. This is the prefix:

```
+//ISO 9070/ANSI::113722::US::CI LABS::
```

Following the prefix is a designation that represents either OpenDoc as a whole, a specific platform, or a specific developer (company name). Therefore, the full ISO string definition for the property kODPropContents, for example, is

```
+//ISO 9070/ANSI::113722::US::CI LABS::OpenDoc:Property:Contents
```

All defined property names and data types should include the OpenDoc ISO string prefix. Value types in a storage unit (such as part kinds) should not.

Creating Properties and Values

If you have a reference to a storage unit, you can add a property to it and then add several values to that property. For the example illustrated in the previous figure, for example, you might use statements such as these (in which the storage unit reference is su):

```
su->AddProperty(ev, kODPropContents);
su->AddValue(ev, kMyKind);
su->AddValue(ev, kKindTEXT);
su->AddValue(ev, kKindPICT);
```

These statements add the property kODPropContents to the storage unit and give it one standard (kODIntlText) and two custom (kCustomTextKind and kMyBitmapKind) value types. The kODPropContents property is a standard property name; you can add your own custom property and value types to a storage unit, using statements such as this:

```
su->AddProperty(ev, kPropAnnotations);
su->AddValue(ev, kMyAnnotationsType);
```

These statements only set up the storage unit to receive data of a particular type; you then need to explicitly store the data in the values. To do so, you must first focus the storage unit, setting it up so that the data you write will be written to the desired value of the desired property.

Focusing a Storage Unit

Because any storage unit can contain a number of properties, and any property a number of values, finding the exact piece of data in an OpenDoc document can be more complex than finding it in a conventional document. OpenDoc allows you to *focus* a storage unit, that is, access the desired data stream (defined by property name and value type) within it before reading and writing.

You focus on a particular stream by calling the storage unit's Focus method and providing some combination of the following three types of specification:

- Name: property name or value type
- Index: the position of the property in the storage unit or the value in the property
- Position code: a code that allows you to specify, for example, the next or previous sibling (a sibling is another value in the same property, or another property in the same storage unit)

If you are not focusing by relative position, you specify a position code of kODPosUndefined when you call the Focus method. For example, all of the following calls would focus on the first value of the contents property of the storage unit shown in the previous figure,

- The names of the property and value:

```
su->Focus(ev,
          kODPropContents,          // Property name
          kODPosUndefined,
          kMyKind,                  // Value type
          0,
          kODPosUndefined);
```

- The name of the property and the indexed position (1-based), within its property, of the value:

```
su->Focus(ev,
          kODPropContents,          // Property name
          kODPosUndefined,
          kODNULL,
          1,                        // Value index
          kODPosUndefined);
```

- The relative positions, compared to the current focus, of the property and value:

```
su->Focus(ev,
          kODPropContents           // Property type
          kODPosUndefined,
          kODNULL,
          0,
          kODPosFirstSib);          // Position code
```

- The storage-unit cursor that specifies the property-and-value pair you want:

```
su->FocusWithCursor(ev, cursor);    // Storage-unit cursor
```

A *storage-unit cursor* is an OpenDoc object that may be convenient if you frequently switch back and forth among specific property-and-value combinations. You can create a number of objects of class ODStorageUnitCursor, initialize them with the storage-unit foci you need, and pass a storage-unit cursor to the Focus method each time you wish to switch focus. You can set up a storage-unit cursor either by explicitly specifying the property and value of its focus, or by having it adopt the current focus of an existing storage unit.

Once you have focused the storage unit, you can read and write its data, as described next.

Manipulating the Data in a Value

To read or write the data of a value, remove data from a value, or insert data into a value, you first focus the storage unit on that particular value.

- To write data at a particular position in a value, call the SetOffset method to set the position of the insertion point in the value's stream, followed by the SetValue method to write the data.

  The SetValue method takes a buffer parameter of type of type ODByteArray; see Handling Byte Arrays and Other Parameters for more information on byte arrays. For example, to write information from the buffer pointed to by dataPtr into the value at the position desiredPosition, you could set up the byte array myData as shown and then call two methods:

```
ODByteArray MyData;

myData._length = dataSize;
myData._maximum = dataSize;
myData._buffer = dataPtr;

SetOffset(ev, desiredPosition);
SetValue(ev, &myData);
```

  To read the data at a particular position in a value, call the SetOffset method, followed by the GetValue method to read the data. To read information from the position desiredPosition in the value into a buffer specified by the byte array myData, you could make these calls:

```
SetOffset(ev, desiredPosition);
GetValue(ev, &myData);
```

  You could then extract the data from the buffer pointed to by myData._buffer.

- To read or write data at the current offset in a value, simply call GetValue or SetValue without first calling SetOffset. specified. To insert data at a particular offset, without overwriting any data already in the value, call the InsertValue method (after having called SetOffset, if necessary).

- To append data at the end of a value, call GetSize to get the size of the value, then call SetOffset to set the mark to the end of the stream, and then call SetValue to write the data into the stream.

- To remove data of a particular size from a particular position in a value, call SetOffset to set the mark to the desired point in the stream, and then call DeleteValue to delete data of a specified length (in bytes) from the stream.

Note: If you change the data in one value of a property, remember that you must make appropriate changes to all other values of that property. All values must be complete and equivalent representations- each according to its own format-of the information the property represents.

Iterating through a Storage Unit

To examine each of the properties of a storage unit in turn, access them in this way:

1. Call the Focus method of the storage unit, passing null values for property name and value type, and kODPosAll for position code. The null values plus kODPosAll "unfocus" the storage unit (that is, focus it on all properties).

2. Get the number of properties in the (unfocused) storage unit by calling its CountProperties method.

3. Focus on each property in turn, by iterating through them using a relative-position method of focusing:

```
su->Focus(ev,
```

```
                kODNULL,
                kODPosNextSib,          // Position code
                kODNULL,
                0,
                kODPosUndefined);
```

To examine in turn each of the values in a property of a storage unit, you access them in a similar way:

1. Focus the storage unit on the desired property by calling its Focus method and passing the desired property name and kODPosAll for relative position of value. Using kODPosAll focuses the storage unit on all values of that property.
2. Get the number of values in the property, by calling the CountValues method of the focused storage unit.
3. Focus on each value in turn, by iterating through them using a relative-position method of focusing for the values, while maintaining the same property position:

```
su->Focus(ev,
                kODNULL,
                kODPosSame,             // Position code
                kODNULL,
                0,
                kODPosNextSib);         // Position code
```

Removing Properties and Values

You can remove a property from a storage unit by:

1. Focusing on the property to be removed; call the Focus method and pass it a specific property name and a value position of kODPosAll.
2. Removing the property, by calling the Remove method on the focused storage unit.

You can remove a value from a property in a storage unit by:

1. Focusing on the value to be removed; call the Focus method and pass it a specific property name and a specific value type.
2. Removing the value; call the Remove method of the focused storage unit.

Storage-Unit IDs

At run-time, the draft object assigns an identifier to each of its storage units. A *storage-unit ID* is a non-persistent designation for a storage unit that is unique within its draft (storage-unit IDs are not unique across drafts and do not persist across sessions). You can use the ID to identify storage units, to compare two storage units for equality at run-time, and to recreate persistent objects from their storage units.

Just as object references are the basic run-time identifiers for OpenDoc objects, storage-unit IDs are the basic run-time identifiers for the storage units of persistent objects (subclasses of ODPersistentObject). Methods that manipulate storage units often take a storage-unit ID as a parameter; ODDraft::AcquireStorageUnit and ODDraft::ReleaseStorageUnit are examples.

For purposes in which the object unit or its storage unit as a whole is passed or copied, an ID is better than a run-time object reference. For example, storage-unit IDs are used when cloning persistent objects (see Persistent References and Cloning). Using a storage-unit ID ensures that the copying occurs even if the storage unit's object is not in memory at the time. However, OpenDoc first looks for the object in memory; if it is there, OpenDoc uses it rather than its storage unit.

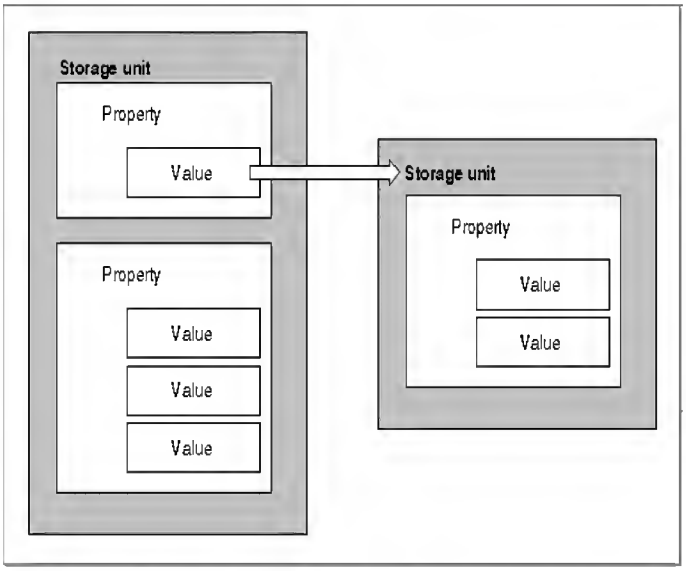
You generally create a persistent object from storage by passing a storage-unit ID to the object's factory method (such as ODDraft::AcquireFrame and ODDraft::AcquirePart). You can also conveniently retrieve persistent objects that may or may not have been purged (such as frames scrolled out of and then back into view) by retaining the storage-unit ID for the object when you release it, and then supplying that ID when you need it again. Note also that, in discussions that refer to part ID or object ID for any persistent object, the ID being referred to is a storage-unit ID.

Storage-unit IDs are not persistent. Therefore, to get the correct storage-unit ID when creating a stored persistent object, a caller must have

access to another, more permanent means of identifying a storage unit. For that purpose, OpenDoc uses persistent references. OpenDoc provides methods (such as ODDraft::GetIDFromStorageUnitRef) for obtaining a storage-unit ID from a persistent reference, and vice versa (such as ODStorageUnit::GetStrongStorageUnitRef).

Persistent References

A *persistent reference* is a number, stored somewhere within a given storage unit, that refers to another storage unit in the same document (see the following figure). The reference is preserved across sessions; if a document is closed and then reopened at another time or even on another machine, the reference is still valid.



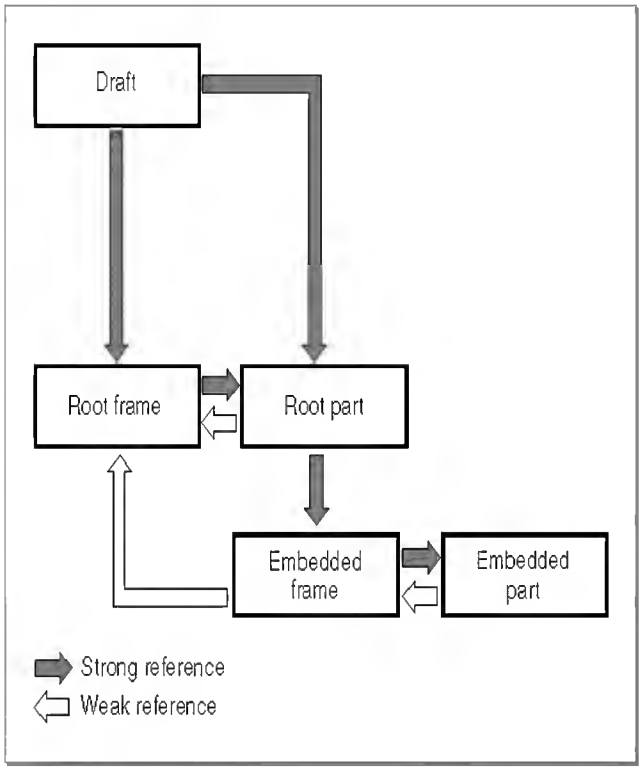
Persistent references allow data to be placed into multiple storage units. The storage units reflect the run-time objects whose data they store; the persistent references permit the reconstruction of the run-time relationships among those objects in subsequent sessions.

Persistent References in OpenDoc

OpenDoc uses persistent references in several situations, including these:

- The stored data in a draft includes a list of persistent references to the root frames of all windows. When a document is opened, the root frames can then be found and their windows reconstructed.
- A stored frame has a persistent reference to its part. When a frame is read into memory, it can then find the part it displays.
- A stored part has persistent references to all of its embedded frames. When a part is read into memory, it can then find all of the embedded frames that it contains. Persistent references to embedded frames are essential to embedding; parts have no access to their embedded parts except through embedded frames.
- A stored part also may have persistent references to all of its display frames. When a part is read into memory, it can then find all of the frames that display it.
- Parts can also use persistent references for hierarchical storage of their own content data.

The following figure is a simplified diagram showing the persistent references among stored objects in an OpenDoc Document. (The difference between the strong and weak persistent references referred to in the figure is explained in Persistent References and Cloning).



Creating Persistent References

To create a persistent reference, you first focus the storage unit on the value whose data stream is to hold the reference and then call the storage unit's GetStrongStorageUnitRef or GetWeakStorageUnitRef method, passing the ID of the storage unit that is to be referred to. You then store the returned reference in the focused value in a format consistent with the type of the value. Such a reference is then said to be from the value to the referenced storage unit.

A persistent reference is a 32-bit value of type ODStorageUnitRef. You can create and store a virtually unlimited number of persistent references in a storage unit value; each persistent reference that you create from a particular value is guaranteed to be unique. Do not try to inspect or interpret a persistent reference; the classes ODStorageUnit and ODStorageUnitView provide methods for manipulating them.

Note: The scope of a persistent reference is limited to the value in which it was originally created and stored. Do not store it in a different value; it will almost certainly no longer refer to the correct storage unit.

Once a persistent reference is no longer needed, you should remove it from the value in which it was written. Extra persistent references threaten the robustness and efficiency of execution. A storage unit is aware of all persistent references that it holds, even those that a part editor may have stored within its contents property. OpenDoc provides an iterator class, ODStorageUnitRefIterator, through which a caller can retrieve all persistent references in a given value.

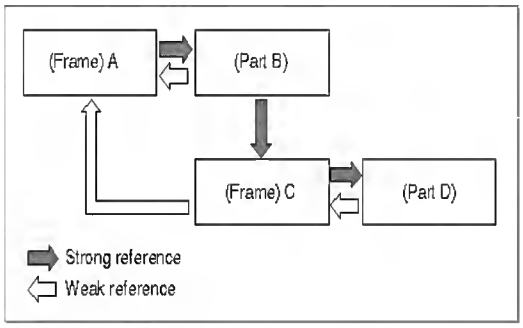
Persistent References and Cloning

There are two kinds of persistent references: strong persistent references and weak persistent references. They are treated differently in cloning.

To *clone* an object is to make a deep copy of it: not only is the object itself copied, but also all objects that it references (plus the objects that they reference, and so on). The cloning process is described further in Cloning.

In a clone operation, copies are made of all storage units referenced with a strong persistent reference in the object being cloned. Storage units referenced with a weak persistent reference are not copied.

The use of weak persistent references allows you to clone only portions of a document or other structure of referenced storage units. The following figure, which shows the persistent references among a stored frame and part plus their embedded frame and part, illustrates how cloning gives different results, depending on which references are strong and which are weak.



For example, if you were to clone the containing frame (A) in the previous figure, all four objects in the figure would be cloned, representing the containing frame and part plus its embedded frame and part; if, however, you were to clone the containing part (B), only the part plus its embedded frame and part (C and D) would be cloned, even though the containing part includes a persistent reference back to its display frame (A). Likewise, if you were to clone the embedded frame (C), only the frame and its part (D) would be cloned.

You define a persistent reference as weak or strong by calling either the GetStrongStorageUnitRef or the GetWeakStorageUnitRef method of the storage unit that is to hold the reference.

Stored display frames
It is recommended, although not necessary, that you store your part's display frames persistently. This will ensure that the part can be closed and will look the same if it is reopened later. If you do store display frames, however, make sure that your part's persistent references to them are weak references. This will enable you to clone the part without its display frame(s), so that the clone can have its own display frame(s).

Main and Auxiliary Storage Units

Every part (or other persistent object) has a single main storage unit, whose kODPropContents property stores the content of that object. In most cases, you can simply write all of your part's data as a stream into that one storage unit. It is possible, however, to create auxiliary storage units that hold additional data related to the data in a main storage unit.

The procedure involves creating a strong persistent reference to the auxiliary storage unit and storing it in your main storage unit. You can subsequently retrieve the reference and convert it into an object reference to the auxiliary storage unit. See Creating Additional Storage Units for more information.

Prefocused Access with Storage-Unit Views

A storage-unit view is an OpenDoc object that represents a prefocused access to a storage unit. The class ODStorageUnitView has most of the functionality of the class ODStorageUnit, except that it has no methods for accessing different properties and values. A storage-unit view does not in itself store data; each storage-unit view is associated with a specific storage unit that actually does the storing. Calls you make to access the storage unit view are passed to its associated storage unit.

Several methods of several OpenDoc classes take a storage-unit view as a parameter. The ODPart methods FulFillPromise, ReadPartInfo, WritePartInfo, ReadActionState, and WriteActionState, for example, all make use of storage-unit views. When one of your methods receives a storage-unit view as a parameter, the method can read from the storage-unit view or write to it without first focusing on any property or value or locking the storage unit.

You can, if desired, create storage-unit views to pass prefocused storage units among your software components, or to any OpenDoc objects (such as ODNameSpace) that you might create, and whose methods take storage-unit views.

Documents, Drafts, and Parts

Compound documents are fundamental to OpenDoc, and each compound document is represented by a document object. Documents are

composed of one or more drafts. This section discusses the relationship of documents to drafts and to parts, describes how part data is stored, and discusses the objects that a document comprises.

The document object is responsible for creating and deleting drafts, for storing the identity of the current draft, and for collapsing multiple drafts into one. Your part editor rarely interacts directly with its document object. However, see Creating a New Document for information on creating a document object.

Drafts

A *draft* is a specific version of an OpenDoc document. Multiple drafts can be maintained within a single document. There is always at least one draft, the *base draft* , within a document. An individual draft can be extracted from a document and placed in a new document for editing.

Drafts are created by users with the help of the document shell, and maintained by the container suite. The figure in section Document Drafts shows an example of the Drafts dialog box, with which the user views and manipulates the drafts of a document. A user can save the current state of the document as a new draft at any time. In general, your part editor can ignore the existence of separate drafts in a document. The data you read or write is always associated with the currently open draft.

Methods of ODDocument allow access to drafts by draft ID or by object reference and relative position in the document. Drafts have a specific set of properties, including creation date, modification date, and user name. Each draft object privately maintains whatever information it needs to distinguish itself from its predecessor draft.

The draft object is responsible for creating and tracking frame objects, part objects, link-related objects, and storage units. It also manages the cloning process and flags changes to the content of any of its parts. Typically, it is to perform these tasks-and not to manipulate a draft as a version of a document-that your part editor interacts with its draft object. Only one user can edit a draft at a time, and only the most recent draft of a given document can be edited. (Drafts cannot be changed if other drafts are based on them). To enforce this, each open draft has an associated set of draft permissions. They specify the class of read/write access that your part editor has to the draft. The following table lists the draft permissions recognized by OpenDoc.

Constant	Explanation
kDPExclusiveWrite	Read access and exclusive-write access
kDPReadOnly	Read-only access
kDPSharedWrite	Read access and shared-write access
kDPTransient	Navigation-only access
kDPNone	No access

Note: The Bento container suite supports only the kODReadOnly and kODExclusiveWrite draft permissions.

When your part initializes itself, or at least before attempting to modify any of its data, it should get the permissions of its draft (by calling the draft's GetPermissions method) and behave accordingly. For example, your part editor should not attempt to make changes to a part when its draft has been opened read-only. Also, certain menu items should be disabled when the user views a read-only draft; see Document Menu and Edit Menu for descriptions.

Storage Model for Parts

Parts, like other persistent objects, have a storage unit in which they can write their state persistently. At the same time, parts are directly involved in many other OpenDoc actions (such as binding, data translation, and data transfer) and thus must satisfy additional storage requirements, so they can share their data with other objects as needed. Here are some of requirements your parts must meet:

- Your part must be able to store multiple representations of its content. Each representation must be a full representation of the content; it cannot contain just that data that differs from the data in another representation.
- A caller must be able to extract at least one representation from your part's storage unit, without having to understand the format of any of the representations.
- A caller must be able to remove all but one representation from your part's storage unit, without having to understand the format of

any of the representations.

- You must store your part's representations in order of *fidelity* , which is the faithfulness of the representation to that of your part editor's native format. Store them in descending order of fidelity, with your part editor's preferred format first.
- It must be possible to distinguish the fundamental content of your part from any annotations to it or information about it. OpenDoc-or any caller with access to your storage unit-can delete these annotations and items of information from memory at any time, without writing them to storage, and therefore you cannot use them to store actual content. You can, however, use them to store non-critical optimizations such as caches.
- Because property names are ISO strings, only simple naming conventions, such as prefixes to establish ownership, are possible. For example, "TC corporation" might use the prefix "TC:" to identify all ISO strings for which it controls the format and meaning. OpenDoc-defined strings have the OpenDoc ISO string prefix; see OpenDoc ISO String Prefix.

To meet these requirements, your part must create a contents property, defined by the property name kODPropContents, in your storage unit. In that property, you need to create-for every representation you wish to store-a value whose type is the part kind of the representation. (Part kind is described in Part Kinds Stored in a Part). Into each of these values, you must write one and only one representation of your part's contents, using the data format appropriate to that part. Order the values by fidelity.

Do not store any part content outside of the contents property. Make sure that all other properties in your part's storage unit are for annotations or extra information, rather than for content. Your part's contents property, however,can include persistent references to other storage units whose contents properties contain additional content data.

To help you decide which data you should store as content and which you should not, consider that content includes any data that the user should be able to save as well as any data that should persist across different part editors, different machines, and different platforms.

This storage model offers greater flexibility than is available for application storage on many platforms' file systems. If your applications use only a single stream to store all of their contents, your equivalent part can simply store everything in a single value in the contents property of its storage unit. If your applications use resources, multifork files, or some other form of structured storage, then your equivalent part can use multiple storage units referenced from a value in the contents property of your part's main storage unit.

Because you can add more storage units at any time and save references to them in any value, you can construct an arbitrarily complex structure for your stored data. Each additional storage unit can have multiple properties, and each property can have multiple values. (Remember that different values within the same property should only be used for different representations of the same data).

What a Draft Contains

The following table shows some of the kinds of information that can be stored persistently as properties of the various objects that make up the storage units in a draft of an OpenDoc document. Note that some of the storage units shown in the following table reflect the objects and persistent references shown in the figure in Persistent References in OpenDoc, as well as the run-time object references shown in Run-Time Object Relationships. Furthermore, some properties should not be accessed by part editors. This list is not complete, nor are all properties shown here required to be present. For more information on these and other standard properties, see "Types, Constants, and Errors" in *OpenDoc Class Reference for the Macintosh* .

Property	Description
Any Storage Unit:	
kODPropObjectType	Type of object stored in this storage unit (draft, frame, part, and so on)
Draft Storage Unit:	
kODPropRootPartSU	Strong persistent reference to the root part of this draft
Frame Storage Unit	
kODPropPart	Strong persistent reference to the part displayed in this frame
kODPropContainingFrame	Weak persistent reference to the containing frame of this frame
kODPropFrameShape	Frame shape of this frame
kODPropPartInfo	Part info (part-specific data) associated with this frame

kODPropPresentation	Presentation of the part displayed in this frame
kODPropInternalTransform	Internal transform of this frame
kODPropFrameGroup	Group ID of the frame group this frame belongs to
kODPropSequenceNumber	Sequence number of this frame in its frame group
kODPropLinkStatus	The link status (in-source, in-destination, or not in link) of this frame
kODPropIsRoot	True if this frame is root frame in window
kODPropIsSubframe	True if this frame is a subframe
kODPropIsOverlaid	True if this frame is overlaid
kODPropDoesPropagateEvents	True if this frame's part propagates events

Part Storage Unit:

kODPropContents	Part content (the actual stored data of this part)
kODPropDisplayFrames	Weak persistent references to the display frames of this part

Clipboard or Drag-and-Drop Storage Unit:

kODPropContents	The contents of the clipboard (or drag-and-drop object)
kODPropSuggestedFrameShape	Suggested shape for frame, if contents are embedded at destination
kODPropLinkSpec	A link specification
kODPropContentFrame	(Exists if data is a single embedded frame)
kODPropProxyContents	Suggested adornments to apply to frame (if data is a single embedded frame)

Link Storage Unit

kODPropLinkSource	Weak persistent reference to the link-source object associated with this link object

Link-Source Storage Unit:

kODPropLink	Weak persistent reference to a link object associated with this link-source object
kODPropSourcePart	Weak persistent reference to the part that contains (or last contained) the source data for this link
kODPropLinkContentSU	Strong persistent reference to the contents storage unit for the linked data
kODPropChangeTime	The date and time of this link's last update
kODPropUpdateID	The update ID for this link's last update

kODPropAutoUpdate	True if link is to be updated automatically

Your part editor is responsible for reading and writing only the data that is stored persistently by your parts; OpenDoc takes care of persistent storage of the other objects listed in this table. Basically, each of the objects can read and write itself.

Other standard frame and part properties are listed in the table in the next section.

Info Properties

Some of the standard properties associated with a part (or, in some cases, its frame) are made visible to the user, either for information purposes only or to allow the user to modify them. This set of properties, called *Info properties* , is displayed in the Document Properties notebook. The last-modified date and time (kODPropModDate) is an example of an Info property that the user cannot change; the name of the part (kODPropName) is an example of an Info property that the user can change. Certain items displayed in the dialog box (such as part category and part size) are not storage-unit properties at all, but are calculated at run-time or obtained from other sourced of information. The following table lists the standard Info properties of parts defined for version 1.0 of OpenDoc.

Property	Explanation
kODPropName	The user-assigned name of the part
kODPropViewType	The view type of the part in the currently selected frame (a frame property)
kODPropIsFrozen	True if the part is bundled, false if not (a frame property)
kODPropIsStationery	True if the part is stationery, false if not
kODPropIconFamily	The icons that represent this part
kODPropPreferredKind	The user-specified part kind for the data of this part
kODPropPreferredEditor	The user-specified preferred editor for editing this part (the ID of the editor that last wrote this part to persistent storage)
kODPropCreateDate	The date and time at which this part was originally created
kODPropModDate	The date and time at which this part was last modified
kODPropModUser	The name of the user that last modified this part
kODPropComments	Comments entered by the user into this part's Document Properties notebook

Your part editor can define and attach additional user properties and store them in the part's content or attach them as properties to its parts' storage units. You can then display them to the user in a Settings dialog box, accessible from the Document Properties notebook. See The Settings Extension for more information.

Container Properties

If your part contains embedded parts, you may want the embedded parts to adopt, by default, some of your current display settings or behavior. For example, if your part is a text part, it may be appropriate for other text parts embedded within it to have the same text characteristics (font, size, style, and so on), unless the user overrides them.

You can define such a set of characteristics as properties and attach them to your parts' storage units. These container properties then become available to embedded parts for adoption. See Adopting Container Properties and Transmitting your Container Properties to Embedded Parts for more information.

Creating a New Document

Under most circumstances, your part editor never creates a document. Other parts of OpenDoc, such as the document shell, handle document creation when the user chooses a menu command such as New or drags a part to the desktop.

If, however, your part editor provides its own document-creation interface to the user, or if it caches its own data in separate OpenDoc documents, then it must use methods of the container suite to create those documents.

In your document creation method, you can follow steps such as these:

1. Create a file object, following whatever platform-specific procedures are required by the file system that is in use.
2. Create an OpenDoc container, document, and draft for the file. Call the session object's CreateFileContainer method, then the container's AcquireDocument method, and then the document's AcquireBaseDraft method.
3. Create the root part. Call the base draft's CreatePart method, then obtain the root part's storage unit and assign it to the draft. Call the draft's Externalize method to save the storage unit.
4. Assign the proper file type to the file. This involves giving it a file type (an OSType) that represents the root part's part kind and assigning the document a creator type ('odtm') that is the signature of the OpenDoc document shell. ('odtm').
5. Release the objects you have created: the root part, the draft, the document, and the container. Finally, delete the platform-specific file object that you created. (This does not delete the file from persistent storage).

Reading and Writing your Part

In OpenDoc, reading an object (also called *internalization*) is the process of initializing (or re-initializing) the object by bringing its stored data into memory from persistent storage. Conversely, writing (also called e*xternalization*) is the process of copying an object's essential data to persistent external storage. In general, when a document is opened, its objects are read into memory. As the user edits the document, the objects are modified. When the user saves the document, the modified objects are written back to storage. (As noted earlier, the container suite controls how data is physically stored; reading and writing may not necessarily involve an immediate transfer of data between memory and an external physical storage medium.)

This section discusses how your part reads and writes its own content data, as well as related OpenDoc objects, such as frames, that it uses.

Your part editor must be able to read and write any parts whose data formats (part kinds) it recognizes. You are not required to read the entire contents of your part into memory at once, and you need not write it all to storage at once, and you need not write it all to storage at once. However, reading must put your part in a state in which it can accept events and handle requests, and writing it must ensure that changes made by the user are not lost.

Your part editor should never change the part kind of any part you handle except as a result of an explicit user request. Any translations required to make a part readable, whether performed by OpenDoc or by your part editor, are first selected by the user. See Binding and Translation for more information.

Initializing and Reading a Part from Storage

A part must be able to read itself (reconstruct itself in memory by reading its stored data). To maximize performance, OpenDoc requires reading and writing only when absolutely necessary. A part is not instantiated and read into memory until its draft's CreatePart or AcquirePart method has been called-meaning that the part editor is needed for tasks such as drawing, editing, frame negotiation, or script execution. When a part is read in, OpenDoc has already bound it to a part editor according to its part kind (and loaded the part editor if it is not already in

memory). Whenever a document in which your part is visible is opened, or whenever your part is added to a document, your part's data must be read into memory. Note that OpenDoc parts should always be ready to work from an empty storage unit as well. The two fundamental methods that your part must implement for initializing itself are InitPart and InitPartFromStorage.

The somInit Method

When your part object is first created (or recreated from storage), and before receiving a call to its InitPart or InitPartFromStorage method, your part receives a call to its System Object Model (SOM) object constructor somInit. The somInit method is inherited from the somObject class of SOM; when you subclass ODPart, you must override somInit.

Your somInit method should initialize (clear) your part object's instance variables. You must not perform any tasks in this method that might fail. You can set pointer variables to null and assign appropriate values to numeric variables. If you have any initialization code that can potentially fail, your part's InitPart or InitPartFromStorage method must handle it.

The InitPart Method

The InitPart method is called only once in your part's lifetime, by your draft's CreatePart method, when your part is first created and has no previously stored data to read. (If your part is created from stationery, this method is never called). This is its interface:

```
void InitPart(in ODStorageUnit storageUnit,
              in ODPart partWrapper);
```

In the simplest case, you could take these steps: Call your inherited InitPart method, which in turn calls its inherited InitPersistentObject method to initialize the information your part needs as a persistent object. (Your classes' initialization methods should always call their superclasses' initialization methods).

1. If your part stores its data in multiple part kinds, add a contents property (type kODPropContents) and a value (of your highest-fidelity part kind) to the storage unit passed to you, thus preparing your part for eventual writing of its data. Add values for other part kinds also, if appropriate. Initialize the values if necessary.

2. If your part stores more than one part kind, add a preferred-kind property (type kODPropPreferredKind) to the storage unit passed to you. Write into a value of that property an ISO string representing your editor's highest-fidelity part kind. Normally, that part kind should equal the type of the first value in your contents property.

3. Save, in a field (such as fSelf) of your part, the contents of the partWrapper parameter passed to this method. It represents an object reference through which OpenDoc interacts with your part. It represents an object reference (pointer) through which OpenDoc calls back to your part. See Part-Wrapper Object for more information.

4. Initialize any data pointers or size values that your part maintains. At this point you might mark your part as clean (unchanged). You could use a "dirty flag" for this purpose, which you initialize as cleared, and then set whenever you change your part's content.

Note: Another approach to InitPart is to defer all writing (such as that in steps 2 and 3) until your part's Externalize method is called.

The InitPartFromStorage Method

The InitPartFromStorage is similar to InitPart, except that InitPartFromStorage also reads in data. Your draft's AcquirePart method calls your InitPartFromStorage method and supplies a storage unit from which the part reads itself. Your part retrieves, from the kODPropContents property of the storage unit, the value (specified by part kind) that represents the data stream you wish to read. This is its interface:

```
void InitPartFromStorage(in ODStorageUnit storage unit,
                         in ODPart partWrapper);
```

In your InitPartFromStorage method, take steps like these:

1. Call your inherited InitPartFromStorage method. That method calls its inherited InitPersistentObject method, which initialize the persistent object information perviously stored for your part.

2. Determine the appropriate part kind of data to read in:

- Your editor may be reading in a part that was created or previously edited by a different editor (see Binding Process). Focus on the kODPropPreferredKind of the storage unit passed to you, and retrieve the part's preferred kind (the part kind you should read, if possible). This part kind may or may not equal the type of the first value in the part's contents property.

- If the kODPropPreferredKind property does not exist or if you cannot read data of that part kind, find the highest-fidelity (earliest in storage order) part that you can read.

  (If there is no preferred kind, take the kind that you can read to be the preferred kind. Use that preferred-kind designation later, when you write to storage.)

If the data is non-OpenDoc file data to which your part editor has just been bound because of a drop operation (see Accepting Non-OpenDoc Data), you can take the following steps. Otherwise, go on to step 3.

- Find, in the contents property, a value type representing the file type. If the item is a text file, for example, OpenDoc will have provided the value type (OpenDoc ISO prefix followed by) "kODKindOS2Text".

- Assuming your part can read data of that file type, you can then obtain the file type and file name from the (ISO prefix plus) kODFileTypeEA and kODFileType values.

- Using this information (such as the file specification), you can then make file-system-specific calls to open the file and process its contents into a buffer in memory. If you must store the data immediately and the draft permissions allow you to, write the data into an appropriate value or values in your part's contents property. Otherwise, you can wait until your Externalize method is called before writing it to your storage unit.

  Skip to Step 7 (you have already read the data into your part).

3. Focus the storage unit on its contents property and on the value containing the part's preferred kind, if you can read that kind. If you cannot read the preferred kind, focus on the highest-fidelity part kind that you can read. (Even if you do not read in the preferred kind at this stage, do not yet change the value in the kODPropPreferredKind property).

4. Read the data into your part's buffer, using the GetValue method.

5. Read in additional objects as needed-that is, as persistent references within your main storage unit's kODPropContents property lead you to information stored elsewhere. See, for example, Creating Additional Storage Units, Storing and Retrieving Embedded Frames, and Reading Linked Content from Storage.

6. If your part's display frames have been stored, read them in as discussed in Storing and Retrieving Display Frames.

7. Call your draft's GetPermissions method and save the results in a field of your part. Respect the permissions in later attempts to write to the draft. See Drafts for more information.

8. Save, in a field (such as fSelf) of your part, the contents of the partWrapper parameter passed to this method. The parameter represents a reference (pointer) through which OpenDoc calls back to your part. See Part-Wrapper Object for more information.

9. Initialize any data pointers or size values that your part maintains. At this time you might, if you maintain a dirty flag, mark your part as clean (unchanged).

Writing a Part to Storage

Your part must be able to write itself (write its data to persistent storage, typically external storage such as a disk) when instructed to do so. This instruction may come at any time, not just when the user saves or closes your document. To maximize performance, OpenDoc requires writing only when absolutely necessary.

- A part is not written to storage until the user performs a save operation on its document, causing the part's Externalize method to be called. If the user does not perform a save operation on a document, no changes to any of its parts are permanently recorded.

- When its document is saved or closed, no writing is required of a part if it has not been altered since it was read into memory.

Your part editor can follow these polices strictly, or it can deviate from them as necessary. For example, you can write out your part at any time, without waiting for the user to save the entire document. Note, however, that no matter how many times your part writes its data to storage, none of those changes will become persistent unless the user performs a save.

This section discusses the Externalize and ExternalizeKinds methods of your part editor. Another method that involves writing your part data to storage is the CloneInto method, described in CloneInto Method of your Part Editor.

The Externalize Method

A caller instructs your part to write its data to storage by calling your part's override of its inherited Externalize method. Your part then writes its data into its storage unit.

As a minimum, your part must write one property: kODPropContents. The kODPropContents property contains your part's intrinsic data. You can write your part's data in multiple part kinds, representing multiple data formats, in the kODPropContents property. Each format is a separate stream, defined as a separate value in the property, and each value must be a complete representation of your part's contents.

You can add other properties and values as annotations to your part's storage unit, and you can access them for your own purposes. However, remember that the OpenDoc binding process looks only at the value types of kODPropContents when assigning an editor to a part.

The fundamental method that your part must override for writing its data to storage is the Externalize method, inherited from the class ODPersistentObject. This is its interface:

```
void Externalize ();
```

In a simple Externalize method, you might take these basic steps:

1. Call the inherited Externalize method to make sure that the appropriate persistent-object information for your part is written to storage.
2. Examine your part to see whether it is dirty-that is, whether the content has been changed since the last write. If it is clean (unchanged), take no further action.
3. Get a reference to your main storage unit by calling your part's inherited GetStorageUnit method.
4. Your editor may be writing a part that was created or previously edited by a different editor (see Binding Process). If so, you need to prepare the contents property of the part to match the part kinds and fidelity order that your part editor uses. (This step is required only the first time you write a part after having been first bound to it).
    - Remove any values in the storage unit that represent kinds that your part editor does not support or does not intend to save.
    - Add values if necessary, so that values for all part kinds that your part editor intends to include are present. Make sure the values are in fidelity order, and make sure that the preferred part kind, determined when you read the part in, is one of the values that you write.
5. Focus the storage unit on the contents property and on each of the part kinds of the data you are writing in turn. Write your data to the storage unit, using its SetValue method. Write one value for each part kind of data you store.

    If your part is a container part, you write persistent references to your embedded frames as part of writing your content. See Storing and Retrieving Embedded Frames.
6. Store references to your display frames, as described in Storing and Retrieving Display Frames.
7. Mark your part as clean (unchanged).

Typically, in writing your content you might write out just two values: one in the preferred part kind, and another in a standard, widely recognized part kind useful for data interchange. If your part editor supports many kinds, you might allow the user to select a single default kind, to avoid the creation of very large files full of redundant data.

Note: Do not change the preferred kind from what it was when you read the part, even if you can easily convert the data to a higher-fidelity part kind. (You can, however, store a higher-fidelity part kind in addition to the preferred kind, if you wish). Such a change is implicit translation, and translation should always be controlled by the user. Maintain the preferred kind until the user instructs you to change it, as in the examples described in Binding with Translation.

Your part can write its contents to its storage unit at any time, not just in response to a call to its Externalize method. However, changes to any parts in a document are actually made persistent only when the user performs an explicit save, and thus only when Externalize is called. If your part updates its storage unit but the user never saves the document, your changes are lost.

The ExternalizeKinds Method

Your part's ExternalizeKinds method can be called whenever your part is expected to save its data with a specific set of formats. For example, OpenDoc calls ExternalizeKinds when the user saves a copy of your part's document in multiple formats . A document-interchange utility or service might call the ExternalizeKinds method of all parts in a document to create a version of the document in which all parts are written in one or more common standard part kinds.

When your part's ExternalizeKinds method is called, it is passed a list of part kinds. This is the method's interface:

```
void ExternalizeKinds (in ODTypeList kindSet);
```

The method should write as many of the specified part kinds as it supports, as well as your part's preferred kind. The method should ignore part kinds on the list that it does not support, and it should remove values (other than the preferred kind) from its storage unit that are not on the list. Just like Externalize, the ExternalizeKinds method should make sure to write the part kinds in proper fidelity order, and not change the preferred kind.

Creating Additional Storage Units

Your part can use persistent references to create an auxiliary storage unit for keeping additional data. The auxiliary unit must be referenced from your part's main storage unit, using a strong persistent reference. In brief, you can follow these steps:

1. Call the CreateStorageUnit method of your part's draft to create the storage unit to hold the data.
2. Focus your part's main storage unit on the value in your contents property that is to contain the persistent reference. (It is not necessary to create a separate value to hold the reference).
3. Create the persistent reference, by calling the GetStrongStorageUnitRef method of your main storage unit.
4. Store the reference anywhere in the value; the only requirement is that you be able to recognize and retrieve the reference later. Write the value to persistent storage, by calling the SetValue method of your main storage unit.

Your part can later retrieve the auxiliary storage unit from the main storage unit, in this way:

1. Focus the main storage unit on the value that contains the persistent reference.
2. Read the value, using the GetValue method of your main storage unit. Retrieve the persistent reference from the value's data.
3. Get the storage unit ID of your auxiliary storage unit from the persistent reference, by calling the GetIDFromStorageUnitRef method of your main storage unit.
4. Get the storage unit itself by passing its ID to the AcquireStorageUnit method of your draft.

How you store the data in your auxiliary storage unit is completely up to you. You can define your own properties, or you can keep the kinds of properties used in your main storage unit. If you define your own property names, avoid using constants for them that start with kOD; only OpenDoc-defined constants should use those characters.

Storing and Retrieving Display Frames

As noted in Working with your Display Frames and Facets, your part should keep a private list of its display frames. It is further recommended that your part persistently store those display frames when it writes itself to storage, and retrieve them when it reads itself from storage. Your part may need information on currently nonvisible, uninstantiated display frames in order to perform frame synchronization (see Synchronizing Display Frames) or frame negotiation (see Frame Negotiation).

If your part object includes a field that is a list of display-frame objects, your InitPart method can first create the field, and your DisplayFrameAdded, DisplayFrameRemoved, DisplayFrameConnected, and DisplayFrameClosed methods can add or delete elements of the list, as appropriate.

You can keep additional information about the frames in this field, such as what portion of your content (like a page number) each displays. That way, you can instantiate only those frames that you currently need to manipulate, using the techniques described in Lazy Instantiation.

You should store your list of frames in an annotation property in your part's storage unit. Your Externalize method should create a property with the name kODPropDisplayFrames in your storage unit and should write the frame list as weak storage unit references into a value of that property.

When reading your part from storage, your InitPartFromStorage method can focus on the kODPropDisplayFrames property and value, and read the list of stored display frames back into your display-frame field. (Be sure your ReleaseAll method releases any remaining display frames in that field; see The ReleaseAll Method).

Storing and Retrieving Embedded Frames

Your part does not explicitly store the data of embedded parts; their own part editors take care of that. You do not even explicitly store the frame objects that you use; OpenDoc takes care of that. You do, however, store persistent references to the frames that are embedded in your part.

The process of storing an embedded frame for your part is simple. All you need to do is store a strong persistent reference to the embedded frame object; OpenDoc takes care of storing the frame itself. You follow the same steps that you take when creating and storing a reference to any storage unit (described in Creating Additional Storage Units):

1. Focus your storage unit on your contents property and the value that is to contain the persistent reference to the embedded frame.
2. Create the persistent reference and place it in your contents data.
3. Write the data back into the value.

When your part is reinstated at a later time, your part can then retrieve the frame this way:

1. Focus the storage unit on the value containing the persistent reference.
2. Retrieve the reference.
3. Get the storage-unit ID of the frame.
4. Recreate the frame itself, by calling your draft's AcquireFrame method.
5. If you have previously stored information regarding a change to the frame's link status (see The LinkStatusChanged Method of your Part Editor), set the frame's link status at this time. Link status is described in Frame Link Status.

For efficient memory use, you can instantiate only those embedded frames that you currently need to manipulate or display, using the techniques described in Lazy Instantiation.

Reading and Writing Part Info

As noted in Part Info, you can use the part info data of a frame or facet to store any display-related information you want to associate with that particular frame or facet. Because frames are persistent objects and facets are not, a frame's part info can be stored persistently, whereas a facet's part info cannot.

Just as you store multiple representations of your part's data, you should store multiple representations of your frames' part info, so that other editors can potentially read your part info as well as your part content. In that case, you should give your part info formats names equivalent to the part kinds with which they are associated with. For example, if you write part data whose part kind is "SurfCorp:SurfWriter:StyledText", your associated part info data should be written in a value whose type is "SurfCorp::SurfWriter:StyledText:PartInfo".

If you also write your part data in a standard format such as PICT or RTF or JPEG, you would not write associated part info for those formats, because they have no associated part info format.

As with other changes to the contents of your part's draft, any time you place or modify information in your display frame's part info field, you should follow the procedures listed in Making Content Changes Known. (If your frame is a nonpersistent frame, however, you do not need to make changes to it known).

Whenever your document is saved, your part's WritePartInfo method is called for each of your part's display frames. Here is its interface:

```
void WritePartInfo(in ODPointer partInfo,
                   in ODStorageUnitView storageUnitView);
```

On receiving this call, you could first examine your part's own part-info dirty flag (if you have defined one) to see if this frame's part info has changed since it was read in from storage. If it has, focus on as many values in the provided storage unit view as is appropriate and write your current part info into them. (And then clear the dirty flag).

Conversely, whenever a display frame of your part is read into memory, your part's ReadPartInfo method is called. Here is its interface:

```
ODPtr ReadPartInfo(in ODFrame frame,
                   in ODStorageUnitViewstorageUnitView);
```

On receiving this call, you should allocate a structure to hold the part info, focus on the appropriate value in the provided storage-unit view, and read the data into your part-info structure. (And initialize your part-info dirty flag to clean).

Your part also writes part info data when its display frames are cloned; see ClonePartInfo Method of your Part Editor for more information.

Changing your Part's Content

Users can change the content of your part through a variety of standard editing actions involving keyboard input, menu commands, scripting, and so on. How you can support those tasks is described throughout this book, with additional information and other programming texts.

This section summarizes the content-changing tasks of adding and removing embedded parts, and it notes the ways in which you must keep OpenDoc and other parts informed when you make content changes of any kind.

Adding an Embedded Part

Embedding is a process that can combine elements of storage, layout, data transfer, menu handling, and control manipulation. This section summarizes the embedding process; the summary is presented in this chapter because embedding involves such a fundamental change to a part's storage configuration.

In terms of the stored persisten objects involved, embedding breaks down to these basic alternatives:

- Your part embeds either a new or preexisting frame into its content.
- The embedded frame displays the content of either a new or preexisting part.

You create a new embedded frame by calling your draft's CreateFrame method; you embed a preexisting frame by calling your draft's AcquireFrame method. In situations where you must explicitly instantiate the part object to embed, you create a new part by calling your draft's CreatePart method, or you obtain an existing part by calling your draft's AcquirePart method.

Detailed procedures for embedding a part within your part are described in detail in Pasting from the Clipboard, Dropping, and Using a Tool Palette to Embed Parts. See those sections for more information. In summary, however, you perform these tasks when you embed:

1. You decide, based on your own part's content model and on information accompanying the transferred data, what area is to be given to the embedded part for display:
    - You retrieve or define a frame shape for the embedded frame. (The part may or may not arrive with a suggested frame or frame shape).
    - You decide where to position the embedded frame in your part's content; based on that position, you will eventually create an external transform to use for positioning the facets of the embedded frame.
2. For each of your own part's display frames that is to show the embedded part, you create or retrieve a part object (if necessary), and then you create or retrieve a frame object.
    - If the part you are embedding is not already attached to a frame, ignore this step. If not, you must manipulate the part itself. Either retrieve the part object (as when embedding pasted content that has no preexisting frame or adding a new frame to an existing embedded part) or create a new part object (as when creating a new part from a palette).

      The embedded part is stored in your part's draft, but not within your part itself.
    - Either retrieve the supplied frame object (as when embedding pasted content that comes with a display frame) or create a new frame object (as when adding a new frame to an existing embedded part or creating a new part from a palette). If you retreived or created a part in the previous step, attach that part to the frame.

      You store a reference to the frame somewhere in your part's content. The frame object itself is stored in your part's draft, but not in your part.
3. If the embedded part's frame is presently visible, you create a new facet (or facets) to display the embedded part.
4. As necessary or appropriate, you perform other tasks related to embedding, such as those described in Working with Embedded Frames and Facets. Finally, you notify OpenDoc and your own containing part that you have changed your part's content.

Removing an Embedded Part

To remove an embedded part from your part, you must remove all the facets of the part's display frame (your embedded frame), release or remove the embedded frame, and redisplay your own part's content. Follow the instructions in Removing an Embedded Frame.

This procedure illustrates the fact that your part is not actually concerned with removing embedded parts themselves. Under user instruction or for other reasons, your part may have reason to remove one or more embedded frames; once the last display frame of a given embedded part is removed from your part, that part is no longer embedded in your part.

Making Content Changes Known

Whenever you make any significant change to your part's content, whether it involves adding, deleting, moving, or otherwise modifying your intrinsic content or your embedded frames, you must make those changes known to OpenDoc and to all containing parts of your part.

- Call your draft's SetChangedFromPrev method. This marks the draft as dirty, giving your part the opportunity to write itself to storage when the draft is closed.
- Call the ContentUpdated method of each of your part's display frames in which the change occurred, passing it an identifying update ID obtained in one of three ways:
    - Returned from a call to the session object's UniqueChangeID method (if the content change originated in your part's intrinsic content)
    - Propagated from a link destination (see Link Update ID)
    - Propagated from a call to your EmbeddedFrameUpdated method (see The EmbeddedFrameUpdated Method of your Part Editor) Calling ContentUpdated notifies your containing parts-and their containing parts, and so on-that your content has changed. The notification is necessary to enable all containing parts to update any link sources that they maintain.

You should call these methods as soon as is appropriate after making any content changes. You may not have to make the calls immediately after every single change; it may be more efficient to wait for a pause in user input, for example, or until the active frame changes. In addition to sending these notifications, you must of course also update your part's display if appropriate; this may involve adding or removing facets, adjusting intrinsic content, and invalidating areas that need to be redrawn.

Closing your Part

When the user closes the document containing your part, or when your part is deleted from its containing part, OpenDoc calls your part's ReleaseAll method, followed by its somUninit method. (If, when closing a document, the user specifies that changes be saved, your part's Externalize method is called before ReleaseAll.)

The ReleaseAll Method

The ReleaseAll method is inherited from ODPersistentObject. Its purpose is to ensure that all your part's references to other objects and ownership of shared resources are relinquished before your part is itself deleted from memory. As a minimum, your override of the ReleaseAll method should:

- Call the Release methods of all reference-counted objects to which your part has references, including iterating through its private lists of embedded frames and display frames and releasing each one (at this point, all of them should have been release anyway)
- Remove any link specifications it has written to the clipboard
- Fulfill any promises it has written to the clipboard
- Relinquish all foci that your part owns
- Clear the undo stack if it has written any undo actions to it
- Call the BaseRemoved method of any of your part's extensions
- Call the PartRemoved method of any embedded-frames iterators your part has created

Your part should not write itself to storage from within its ReleaseAll method. Your part's Release method, inherited from ODRefCntObject, is called under different circumstances from ReleaseAll. For information on implementing a Release method, see Reference-Counted Objects.

The somUninit Method

After it completes ReleaseAll, your part receives no subsequent method calls except to its System Object Model (SOM) object destructor somUninit. The somUninit method is inherited from the somObject class of SOM; when you subclass ODPart, you must override somUninit.

Your somUninit method should dispose of any storage created for your part object by the somInit method and any other storage related to additional instance variables of your part initialized during execution. In this method, do not perform any tasks that could fail.

Data Transfer

This is the sixth of eight chapters that discuss the OpenDoc programming interface in detail. It describes how OpenDoc provides support for data-transfer operations.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. You should also be familiar with OpenDoc storage concepts, as presented in the previous chapter. For additional concepts related to your part editor's run-time environment, see OpenDoc Run-Time Features.

This chapter describes general issues common to all data-transfer methods, and then describes how your part can support

- Clipboard data transfer
- Drag and drop data transfer
- Linking

Storage Issues for Data Transfer

This section introduces some of the storage-related concepts used in clipboard transfer, drag and drop, and linking, which are described in detail later in this chapter.

For the purposes of this section, the term data-transfer object means any of the following: the clipboard or drag and drop object (when reading or writing data), a link-source object (when writing linked data), or a link object (when reading linked data).

Data Configuration

When a part places data on the clipboard or other data-transfer object, it writes a portion or all of its intrinsic content to the data-transfer object, and it also can cause the contents of one or more embedded parts to be written.

Whatever the nature of the data, once it is written to the data-transfer object it can be considered an independent part. It has its own intrinsic content and it may contain embedded parts. In writing or reading the data, your part is directly concerned only with the characteristics (such as part kind) of the intrinsic content of the transferred part. Any embedded parts are transferred unchanged-as imbedded parts-during the process.

The storage unit for the data-transfer object's intrinsic content is called its *content storage unit* . It is the main storage unit for the content of the data-transfer object, equivalent to a part's main storage unit (see Main and Auxiliary Storage Units). Any embedded parts in the transferred data, and any other OpenDoc objects, have their own storage units.

Note: A link object or a link-source object is a persistent object, and as such has its own main storage unit. That storage unit is not the same as its content storage unit. Always access the content of a data-transfer object by calling its GetContentStorageUnit method.

More specifically, the data-transfer object's intrinsic content is stored in the contents property (type kODPropContents) of the content storage

unit. When writing to a data-transfer object, your part can store data in multiple formats in different values of the contents property. Just as with a part, each value in the contents property must be complete; it must not depend on other properties or other values of the property.

Your part accesses other properties besides the contents property in a data-transfer object. For example, when your part writes data to a data-transfer object, it may also write-into a separate property-a frame object or a frame shape for the destination part to use when constructing a display frame. See Annotations for more information.

Objects in memory take precedence over their stored versions in data transfer. If the user cuts or copies a part's frame to the clipboard or other data-transfer object, the moved data represents the current state of the part, including any edits that have not yet been saved to disk.

Annotations

This section discusses the items, in addition to part content, that you can write to or read from the storage unit of a data-transfer object.

Link Specification

When copying content to the clipboard or drag and drop object, a part advertises the ability to create a link by writing a link specification in addition to content. When you copy data to the clipboard or drag and drop object, you should create a link specification-using your draft's CreateLinkSpec method-in case the user chooses to link to the data when pasting or dropping it in the destination. Writing Intrinsic Content. illustrate when to write a link specification.

Write the link specification onto the content storage unit of the clipboard or drag and drop object as a property with the name kODPropLinkSpec. The data in a link specification is private to the part writing it. The data you place in your link specification is returned to your part if and when your part's CreateLink method is called to create the link. All that your part needs from the link specification data is sufficient information to identify the selected content.

Because the link specification is valid only for the duration of the current instantiation of your part, the link specification can contain pointers to information that you maintain.

Link specifications are not necessary in the following situations:

- When you place content on the clipboard as a result of a cut operation. You cannot link data that is no longer in your part. (Because you cannot know at the start whether a drag will be a move or a copy, you should always write a link specification when you write data to the drag and drop object.)
- When you write all or a portion of a link destination (or any of your part's content, if your part itself is in a link destination) to the clipboard or drag and drop object. Creating a link in that situation could make your link destination a source of another link. Tha configuration is technically possible but generally a bad practice. (see the second table in Transfer Rules for Links and Link Sources for an explanation).
- When your draft permissions (see Drafts) are read-only. You would not be able to establish links to other parts of the same (read-only) document, and any links to other documents would be broken when your draft closed.
- When you are writing to a link-source object. Link specifications are for the clipboard or drag and drop object only. Link specifications are transitory, whereas links are persistent.

When you write a link specification to the clipboard, obtain and save the clipboard update ID (see Clipboard Update ID). You must remove a link specification from the clipboard if your source data changes so that the clipboard data no longer reflects it, and you need the update ID to test for that situation. See Removing a Link Specification from the Clipboard.

(You never need to remove a link specification from the drag and drop object, because it is valid only during the course of a single drag operation.)

Note: The kODPropLinkSpec property is not copied when the storage unit containing it is cloned.

Frame Shape or Frame Annotation

When you place intrinsic content (with or without embedded frames) on the clipboard or other data-transfer object, there is no frame associated with that content. You should, nevertheless, write a frame shape to the data-transfer object to accompany the content; the shape is

a suggestion to any part that reads the data and must embed it as a separate part. You write the frame shape into a property named kODPropSuggestedFrameShape in the data-transfer object's content storage unit.

Likewise, if your part receives a paste or drop from the data-transfer object and embeds the instrinsic data, your part should examine the kODPropSuggestedFrameShape property to get the source part's suggested frame shape for the data. If the kODPropFrameShape property does not exist, use your own part-specific default shape.

If you place one of your embedded parts (in the form of a single embedded frame) on the clipboard or other data-transfer object, you must place the frame object there also. You write the frame into a property with the name kODPropContentFrame in a storage unit of the data-transfer object's draft.

Likewise, if your part receives a paste or drop from the data-transfer object, you can note from the presence of the kODPropContentFrame property that the data represents a single frame that should be embedded, and you can retrieve the frame from the property.

When you transfer a single embedded frame, you can specify the frame location relative to your part's content (that is, the offset specified in the external transform of the frame's facet) by incorporating the offset into the frame shape that you write. Then, the receiving part of the paste or drop can, if appropriate to its content model, use that offset to construct an external transform.

Note: Neither the kODPropSuggestedFrameShape nor kODPropContentFrame property is copied when the storage unit containing it is cloned.

Proxy Content

When you write a single embedded frame to a data-transfer object, you can optionally write any intrinsic data that you want to associate with the frame. The intrinsic data might be a drop shadow or other visual adornment for the frame, or it might be information needed to reconstruct the frame as a link source or link destination.

This information is called proxy content; to write it, you add a the kODPropProxyContents property to the data-transfer object, and write your data into it as a value that your part recognizes. If the transferred part is subsequently embedded into a part that also recognizes that value and knows how to interpret it, the added frame characteristics can be duplicated.

Likewise, if your part receives a paste or drop and embeds the single part, you can note from the presence of the kODPropProxyContents property that proxy content for that frame exists. If your part understands the format of the proxy content-which you should be able to determine by examining the value types in the property-you can read it and duplicate the frame characteristics.

Note: The kODPropProxyContents property is not copied when the storage unit containing it is cloned.

Cloning-Kind Annotation

When a single embedded part is written to a data-transfer object, its containing part writes a property with the name kODPropCloneKindUsed to the data-transfer object. The property specifies the kind of cloning transaction used to clone the embedded part. If the embedded part writes a promise instead of actual data to the data-transfer object, it uses the information in the kODPropCloneKindUsed property when it later fulfills the promise.

For more information, see Fulfilling a Promise.

Mouse-Down Offset Annotation

If your part initiates a drag operation (see Initiating a Drag), you need to create a property named kODPropMouseDownOffset in the drag and drop object's storage unit. Write into that property a value that specifies the location of the mouse-down event that triggered the drag. The value should be of type ODPoint and should contain the offset from the origin of the content being dragged.

If your part receives a drop, it should likewise check for the presence of the kODPropMouseDownOffset property. If the property exists, and if taking it into account is consistent with your part's content model, use it to locate the dropped content in relation to the mouse-up event that marks the drop.

Note: The kODPropMouseDownOffset property is not copied when the storage unit containing it is cloned.

Drag and Drop Annotations (OS/2)

Each content storage unit of the drag and drop object contains a PM DRAGITEM structure corresponding to a drag item in the current drag and drop operation. It is written by the drag and drop object under the kODDragitem value of the kODPropContents property. Usually parts don't have to interact with this value other than in their Drop method, when they have to create a storage unit view for it and pass it to the GetDataFromDragManager method of the drag and drop object to perform data rendering.

The drag and drop object supports data rendering for two mechanisms: DRM_SHAREDMEM and DRM_OS2FILE. If either one of these mechanisms has been selected (RMF selection is done by calling drag and drop CanEmbed or CanIncorporate methods from within the part's DragEnter method), the storage unit rendered by GetDataFromDragManager will contain the rendered data with no kODDragitem value. If the selected rendering mechanism is DRM_OS2FILE and this is not an OPENDOC initiated drag, the rendered storage unit will contain the following values for its kODPropContents property:

Selected kind	Part kind that the selected RMF was mapped to; no value is set for this kind. The actual data has to be read from the dropped file.
kODFileType	Name of the dropped file; value type is ODISOStr.
kODFileTypeEA	Any extended attributes that are associated with the file; value type is ODISOStr.

If the selected rendering mechanism is not handled by the drag and drop object, the target part has to perform the data rendering. In this case, the kODPropContents in the storage unit rendered by GetDataFromDragManager will contain a set of values necessary for the part to carry out the rendering conversation:

kODDragitem	DRAGITEM structure; value type is DRAGITEM.
kODSelectedRMF	Selected rendering mechanism and format pair; value type is ODISOStr.
kODSelectedKind	Part kind that the selected RMF was mapped to; value type is ODISOStr.
kODDragOperation	PM drag operation from the DRAGINFO structure associated with this drag; value type is ODUShort.

Clonable Data Annotation Prefix

In some situations an entity may need to store properties in the storage unit of a part or other object without the knowledge or cooperation of the part itself. For example, a service such as a spell checker might store a dictionary of exceptions as a property of a part's storage unit. The part is unaware of the existence of that property, but the spell checker would want the dictionary cloned whenever the part is cloned.

When a storage unit itself is cloned (see Cloning), all its properties are copied, no matter who wrote them into the storage unit. However, the in-memory verstion of an object is given preference over its storage unit during cloning, because recent, unsaved changes to the object should be included in the cloning operation. Unfortunately, when an in-memory object clones itself, any of its properties that the object itself is unaware of are not cloned, because it does not know to write them into the destination storage unit.

To support this capability, To get around this difficulty, OpenDoc defines the property prefix constant kODPropPreAnnotation, whose value is OpenDoc:Annotation:. When that prefix is part of a property name (for example, OpenDoc:Annotation:Exceptions) in an object's storage unit, OpenDoc always copies that property when the object is cloned, even if the object being cloned is in memory and regardless of whether the object is aware of the existence of the property.

Therefore, if your part stores data in another object that the object itself does not use but that you want to be cloned along with the object, make sure you store it in a property whose name starts with the string defined by kODPropPreAnnotation.

Cloning

You should always transfer data to and from the clipboard or other data-transfer object in the context of cloning. This section describes how cloning works, what the scope of a cloning operation is, and how to implement your part editor's CloneInto method.

To clone an object is to copy the object itself as well as all objects that it references, plus any objects that those objects reference, and so on. Typically, copies are made of all storage units-or their equivalent, instantiated persistent objects-that are referenced with strong persistent references, starting with the object being cloned. Storage units referenced only with weak persistent references are not copied. For more information on how strong and weak persistent references affect cloning, see Persistent References and the figure in section Persistent References and Cloning.

Actually, each object that is cloned during the operation decides-if it is in memory at the time-which of its own storage-unit's properties and which of its referenced objects should be included. If the object is not currently instantiated, all of its storage unit's properties and all of its strongly referenced objects are copied.

Cloning Sequence

All persistent objects have a CloneInto method by which they clone themselves, but your part editor should not call the CloneInto method of any object directly. Instead, you clone an object by calling the Clone (or WeakClone) method of its draft object. The Clone method in turn calls the CloneInto method of the object involved. (Your parts, as persistent objects, must provide a CloneInto method. See CloneInto Method of your Part Editor for more information.)

Cloning is a multistep transaction, designed so that it can be terminated cleanly if it fails at any point. You perform a clone by calling three methods, in this order:

BeginClone

First, call the BeginClone method of the draft object of the data to be cloned. If you are transferring data from your part, call your part's draft object; if you are transferring data from a data-transfer object, call that object's draft object. BeginClone sets up the cloning transaction.

(If you are cloning as a result of pasting or dropping data into your part, you must also specify the destination frame, the display frame of your part that is receiving the paste or drop.)

When you call BeginClone, you are returned a draft key, a number that identifies that specific cloning transaction. You pass that key to the other cloning methods that you call during the transaction. You also specify in the kind parameter to the BeginClone method the kind of cloning operation that is to be performed, so that OpenDoc can maintain the proper behavior for linked data that is being transferred. The following table lists the kinds of cloning transactions that OpenDoc recognizes. Cloning explains how these different types of transactions result in different behavior for links. The following table lists the kinds of cloning operations recognized by OpenDoc.

kind parameter	Meaning
kODCloneCopy	Copy object from source to data-transfer object
kODCloneCut	Cut object from source to data-transfer object
kODClonePaste	Paste object from data-transfer object to destination
kODCloneDropCopy	Copy object to the destination of a drop
kODCloneDropMove	Move object to the destination of a drop
kODCloneToLink	Copy object from source to update a link source
kODCloneFromLink	Copy object from link to update a destination
kODCloneAll	Clone all objects (your part should not use this)

Even when transferring only intrinsic content (and not actually cloning any objects), you should still bracket your transfer with calls to BeginClone and EndClone. That way, you notify OpenDoc of the kind of operation (such as cut or copy) that is being performed and you ensure that the right actions occur at both the source and destination of the transfer.

Clone

For each object that you are cloning, call the draft's Clone method. Clone allows the draft object to specify and recursively locate all objects that are to be cloned. It calls the CloneInto method of the object to be copied, which results in calls to the CloneInto methods of all referenced objects, and so on. For example, when Clone calls the CloneInto method of a part, the part clones its embedded frames; the embedded frames, in turn, clone the parts they display, and so on.

(You sometimes call the draft's WeakClone method instead of Clone, especially when you are cloning within the context of your own CloneInto method. See CloneInto Method of your Part Editor for more information.)

Take these steps when calling the Clone method:

1. First, obtain an object ID to pass to Clone.
    - If you are cloning from a data-transfer object into your draft, make sure that the object you are cloning is valid. Starting with the persistent reference that specifies the object to be cloned, call the IsValidStorageUnitRef method of the storage unit or storage unit view that contains the persistent reference. Never assume that a persistent reference is valid.

      Then, get the object's ID (See Storage-Unit IDs) from the persistent reference by calling the GetIDFromStorageUnitRef method of the storage unit.
    - If you are cloning from your draft into a data-transfer object, your access to the objects to be cloned may be through object references instead of persistent references. In that case, get an object ID by calling the referenced object's GetID method.
2. Pass the object ID to the Clone method.
3. Save the resultant object ID that Clone returns, along with the IDs returned from other calls to Clone, until cloning is complete and you have called EndClone (See EndClone).

   (If you are not actually cloning objects but simply reading or writing intrinsic data, this is the point at which to read or write, instead of calling Clone.)

In cloning, the in-memory version of an object takes precedence over its stored version. For this reason, an object does not need to be written to storage prior to being cloned. If the object is in memory, its CloneInto method is called to perform the clone; if the object is not in memory, its storage unit performs the clone operation.

This convention also means that, if an object is in memory, properties attached to its storage unit that the object itself does not know about might not be copied, unless they are specially named; see Clonable Data Annotation Prefix for an explanation.

Null IDs when cloning links
The Clone method returns a value of kODNullID if the desired object cannot be cloned. For example, Clone does not allow you to clone a link object or link-source object into a link, and it returns kODNullID if you attempt to do so. In this case, because you cannot clone the object, you should delete any data associated with it that you have written into the data-transfer object. However, in the case of a link or link source you should still write the content formerly associated with the object, but as unlinked content.

Annotation properties not cloned
When data is cloned from a data-transfer object, most of the annotation properties (such as kODPropLinkSpec) that the source part may have added to the content storage unit are not transferred, because they make no sense as properties outside of the data-transfer object. If you clone a storage unit from a data-transfer object and need these properties, you must read them from the data-transfer object's storage unit, not from the cloned storage unit.

EndClone

Finally, call the draft's EndClone method. EndClone commits to and actually performs the cloning operation. After EndClone completes, you can then use or reconnect the cloned objects.

1. Pass each returned object ID that you have saved to your draft's IsValidID method to determine if the object was cloned.

2. If IsValidID returns true, you can at this point reconstruct either the cloned object in memory or a persistent reference to it:

    - If you are cloning from a data-transfer object into yur draft and IsValidID returns true, call your draft's AcquireObject method to read in the object and obtain a reference to it.
    - If you are cloning from your draft into a data-transfer object and if IsValidID returns true, call the GetStringStorageUnitRef or GetWeakStorageUnitRef method of the cloned storage unit, as appropriate.

3. If IsValidID returns false, the object was not cloned and you should exclude it from the data you are reading or writing.

    If, at any time after calling BeginClone, the operation cannot be completed, you can terminate the transaction by calling the AbortClone method instead of EndClone.

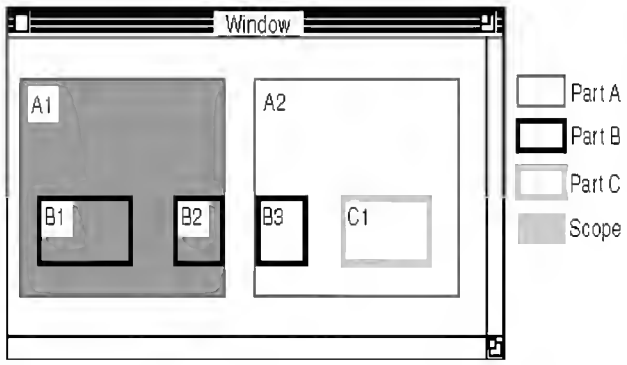
Note: You cannot instantiate and use any cloned object until after the entire cloning operation is complete. If you are cloning several parts and link objects, for example, you cannot call AcquirePart or AcquireLink until after you have cloned all of the objects and EndClone has successfully returned.

Scope of a Clone Operation

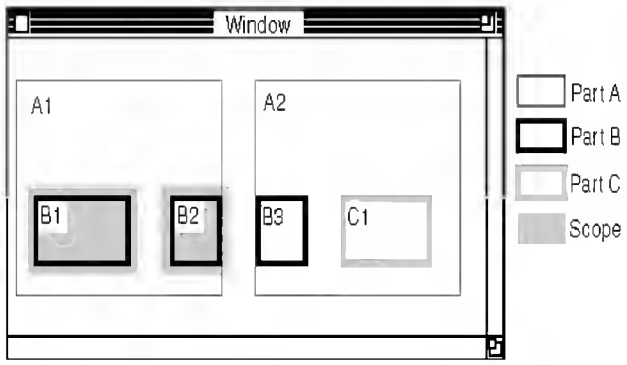
For cloning, the scope defines the set of objects that are to be included in the cloning operation. Scope is expressed in terms of a frame object or its storage unit.

Because a part can have more than one display frame, and because each frame can include a separate set of embedded frames and parts, it is important to specify the frame whose enclosed objects are to be cloned. Otherwise, extra embedded parts or other objects not needed by the copy may be included unnecessarily.. (You can specify null for the scope of a clone if you want all objects copied, regardless of what display frame they are associated with.)

In the example shown in the following figure, the user has selected some content in the root part that includes display frame 1 of embedded part A. The root part writes its intrinsic content and then clones part A, passing it a scope of frame A1. Any content that belongs only to frame A2 (such as part C) is not included in the clone.



Scope changes during the course of a clone. Continuing the example shown in the previous figure, the next figure shows how part A clones itself in the context of the scope (frame A1) specified by the root part. Part A writes the intrinsic content of its display frame A1 and then clones part B twice, first passing it a scope of frame B1 and then a scope of frame B2. Part B thus gets called to clone itself twice, with different scopes. Any content of B within frames B1 and B2 is included, but any content that belongs only to frame B3 is not.



CloneInto Method of your Part Editor

If your part is an embedded part that is written to the clipboard (or other data-transfer object), your part's override of its inherited CloneInto method is called by your draft's Clone method. This is the interface to CloneInto:

```
void CloneInto(in ODDraftKey key,
               in ODStorageUnit toSU,
               in ODFrame scope);
```

Your CloneInto method is passed a draft key for the clone operation and a frame that defines the scope of the cloning. The method should write your part's intrinsic content to the provided storage unit, and it should clone in turn any objects (such as embedded frames or auxiliary storage units) that it references. It needn't clone any objects or write any data that is outside of the scope.

Note: Do not implement your CloneInto method by writing your part to storage and then cloning your storage unit. Doing so would levy performance penalties because of the extra time needed to store your data. Also, it could result in the copying of unneeded objects because the scope of the clone would be ignored.

To support efficient data transfer, your part should, if possible, write a *promise* (see next section) instead of writing its actual intrinsic data when CloneInto is called. It is possible to write a promise only when your part is placed into the data-transfer object as a single, standalone frame with no surrounding intrinsic content of its containing part. In any other situation, your CloneInto method might have been called to help fulfill a promise, in which case writing a promise would be inappropriate. You can determine whether you can write a promise by examining the provided storage unit. If it contains a property with the name kODPropContentFrame, your part is the sole content of the storage unit's contents property, and you can write a promise instead of actual data.

When you write a promise, be sure to identify (to yourself) the display frame or frames of your part that are within the scope of the clone operation, so that your FulfillPromise method can write the proper content when it is called.

Take these general steps in your CloneInto method:

1. Check to see if your part has already been cloned in this operation. Because an object can be referenced more than once, its CloneInto method can be called more than once in a single cloning. In general, if a contents property already exists in the storage unit passed in the method, your part has already been cloned and there is no need to repeat the process.

   An exception to this rule occurs when scope is significant. If your part is called to clone itself with different scopes during the same operation (see, for example, the previous figure), it may have to write additional data each time its CloneInto method is called.

2. Call your inherited CloneInto method.

3. If it does not already exist, add a property named kODPropContents to the provided storage unit. (You do not have to call BeginClone; that method will already have been called. Also, you do not have to add properties other than the contents property; OpenDoc will add your part's name and any other needed annotations.)

4. Check whether you can write a promise. Look for a property with the name kODPropContentFrame in the storage unit. (It is not required that any value yet exist in the property; the caller of your CloneInto method may write the value after cloning completes.)

5. Focus on the contents property, and write a value for each part kind you support. Either write a promise or the data of your part itself, using either the storage unit's SetPromiseValue or SetValue method, respectively.

   (For data transfer, it is especially important to write a standard format in addition to your own native part kind, because the ultimate destination of the transferred data is unknown.)

6. Clone all objects that your part references (that are within the scope of the cloning operation) as follows:

   - For each object to which your part has a strong persistent reference, call your draft's Clone method to clone the object, passing the same draft key that was passed to your CloneInto method. Pass a new scope, if appropriate; for example, if you are cloning an embedded part (and thus its part), that frame is the new scope.

   - For each object to which your part has a weak persistent reference, call your draft's WeakClone method, passing the appropriate draft key and scope.

     (Calling WeakClone does not by itself cause an object to be copied; it only ensures that, if the object ends up being copied because of an existing strong persistent reference to it, your part's weak persistent reference will be maintained across the cloning operation.)

ClonePartInfo Method of your Part Editor

Whenever your part's display frame is cloned during data transfer, the frame calls your part's ClonePartInfo method:

```
void ClonePartInfo(in ODDraftKey key,
                           in ODType partInfo,
                           in ODStorageUnitView,
                           in ODFrame scope);
```

You should respond to this method call by writing your part info into the provided storage unit view (regardless of the state of your part-info dirty flag), and cloning any objects referenced in your part info data.

Promises

Clipboard transfer, drag and drop, and linking can make use of promises. A promise is a specification of data to be transferred at a future time. If a data transfer involves a very large amount of data, the source part can choose to write a promise instead of actually writing the data to a storage unit. When another part retrieves the data, the source part must then fulfill the promise. The destination part does not even know that a promise is being fulfilled; it simply accepts the data as usual.

The format of a promise is completely determined by the source part. The only restriction is that the promise must be able to be written to a storage-unit value.

You are encouraged to write promises in place of actual data in most cases; it minimizes memory requirements and increases performance.

Writing a Promise

Your part can follow these steps to write a promise when it is the source of a data transfer-that is, when it responds to a mouse-down event that initiates a drag or when it copies data to the clipboard, or when it updates a link source.

1. Gain access to the storage unit of the data-transfer object. See, for example, the initial steps under Copying or Cutting to the Clipboard, Initiating a Drag, and Updating a Link at the Source. Focus the storage unit to the contents property (kODPropContents).

2. Prepare your promise. It can have any content and format you decide; you must be able to read it later and reconstruct the exact data that is to be transferred.

3. Write the promise into a single value of the storage unit, using the storage unit's SetPromiseValue method. Your promise must include at least this information:

    - A specification of the actual content that is to be delivered later.
    - A specification of the display frame (or frames) of your part involved in the data transfer. When you fulfill the promise, you can then supply the proper scope for the cloning operation.
    - A specification of the proper kind of cloning transaction (such as kODCloneCopy or kODCloneCut) to apply when you fulfill the promise.

4. Write other needed information. You need not at this stage clone any frames or read any actual content into the data-transfer object. However, you should-as ususal-create a link specification, obtain an update ID, write a suggested frame shape, and so on, as described in Writing Intrinsic Content.

Each promise you write is for a single part kind. You can write several promises representing data of several kinds, so that the destination part has a better chance of being able to incorporate the data instead of embedding it. (Because promises are private data, the actual content of all your promises can be the same, regardless of the part kind being promised. When you are called to fulfill the promise, you can inspect the provided storage-unit view object to find out which part kind is needed.)

Because a promise is valid only for the duration of the current instantiation of your part, the promise can contain pointers to information that you maintain.

Getting Data from a Value Containing a Promise

When the Drop method of a destination part retrieves the data from a drop, or when a destination part reads data from the clipboard (using the GetValue or GetSize methods of the dragged data's or clipboard's storage unit), the source part is called to fulfill the promise. The destination part does not even know that a promise is being fulfilled; it follows the procedures described in Reading from a Data-Transfer Object and uses the same code whether the value contains a promise or not.

Fulfilling a Promise

OpenDoc calls your source part's FulfillPromise method when a promise must be fulfilled, passing it a storage-unit view object that contains the data of the promise. This is its interface:

```
void FulfillPromise(in ODStorageUnitView promiseSUView);
```

In your implementation of the method, take steps similar to these:

1. Examine the private information that you wrote into the promise earlier, to determine what data to write.
2. Look for a property named kODPropCloneKindUsed in the storage-unit view passed to you. Your containing part may have placed the property there if your part was being cloned as a single embedded frame when your part wrote the promise. If the property exists, it contains a value of type kODCloneKind that specifies the kind of clonting transaction (such as kODCloneCopy or kODCloneCut) that you must use in fulfilling the promise.

   If the property does not exist, use information that you saved when you wrote the promise to determine the kind of clonging transaction to use.
3. Begin a cloning operation (call to BeginClone).
4. Retrieve the actual data that the promise represents and write it into the provided storage-unit view object, following the steps described in Cloning Sequence. You can clone frames and other objects as usual at this time.
5. End the cloning operation (call EndClone).

When your FulfillPromise method completes, the destination part receives the data.

When you fulfill a promise, you must be sure to supply the source content that was selected at the time the promise was written, even if that content no longer exists in your part.

If fulfilling your promise requires cloning, you must specify the scope and the appropriate kind of cloning transaction. If you have saved that information in the promise itself, extract it and pass it to your draft's BeginClone and Clone methods, respectively.

There are some times when your part may have to fulfill a promise on its own, without its FulfillPromise method having been called. For example, when your part closes, your part's ReleaseAll method (see Closing your Part) must fulfill all outstanding promises. To fulfill a promise in that manner, your part needs to have kept a record of the promises (and their update IDs) that it has written. Then, in ReleaseAll, it can:

- Access the clipboard content storage unit and verify the clipboard update ID against the part's stored update ID
- Access each value that it has written and verify that it is a promise by calling the storage unit's IsPromiseValue method
- Extract the promise from the value, fulfill it, and write the fulfilled data back into the value.

You can also force fulfillment of one of your promises by focusing on the promised value and calling the GetSize method of its storage unit. That causes your FulFillPromise method to be called immediately.

Translation

The OpenDoc translation object, implemented on each platform as a subclass of ODTranslation, is a wrapper for platform-specific translation services. OpenDoc and part editors can use the translation object to convert part data from one format (part kind) to another.

Through the translation object, OpenDoc maintains information on what kinds of translations are available on the user's system. OpenDoc and part editors can then use the translation object to perform any requested translations, rather than directly calling the platform-specific translation service itself.

In all cases, translation should occur only with explicit user approval and instruction. OpenDoc initiates translation in the situations described in Binding with Translation. Your part editor can initiate translation in the following situations:

- When embedding or incorporating parts through clipboard paste or through drag and drop, your part editor performs translation if the user specifies-in the Paste As dialog box-a part kind that requires translation. See Handling the Paste As Dialog Box for more information.
- Your part editor performs translation when updating a link destination for which the user specified translation in the Paste As dialog box when originally creating the link.

The user specifies the specific translation to perform by selecting a new part kind for the part in the kind pop-up menu of the Paste As dialog box. The kind pop-up menu allows the user to select a part kind to translate to.

To set up the information in the pop-up menu, OpenDoc examines each part kind in the part and determines from the translation object what new part kinds (supported by available editors) the original part kind can be translated to. OpenDoc then presents those choices to the user.

Once the user selects a part kind, your part editor calls the translation object to perform the translation. It is possible for the user to request translation to a part kind that your part editor cannot read. In such a case, you perform the translation but embed the data as a part, and another editor (also chosen by the user) is then bound to the data that you have translated.

The translation object allows only one-step translation; conversion can be only from the existing part kind directly to the part kind selected by the user. Note also that translation applies only to the outermost (intrinsic) portion of the data; parts embedded within it are not translated.

For detailed procedures to follow in translating transferred data, see Translating Before Incorporating or Embedding.

Handling Cut Data

The user can remove data from your part by selecting the Cut command or by using drag and drop to move (rather than copy) data from your part into another part. In this event, you need to take extra steps to account for the fact that the data is still valid but is no longer in your part, and also to allow the action to be undone. Keep these points in mind:

- Cutting must be an undoable action. If the data you cut from your part includes references to objects such as embedded frames or link objects, you should retain those references in an undo action (see Adding an Action to the Undo History). However, your part's content should release its references to the objects. Set the in-limbo flag of each cut embedded frame appropriately (see the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames).

  Once the objects are no longer needed for undo support, OpenDoc calls your DisposeActionState method. You should then either release them entirely or remove them from your draft, depending on whether other valid references to them remain. (Follow the guidelines listed in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames.) See also Undo for Clipboard and Undo for Drag and Drop for related information.
- When you cut an embedded frame to a data-transfer object, call the SetContainingFrame method of the cut frame, setting its containing frame to null and severing it from your part's embedding hierarchy.
- Remove the facets of the cut frame. To access the facets of a frame that the user has drag-moved out of your part, use the embedded-facet iterator of your own display facet. For each facet to be removed, call its containing factet's RemoveFacet method and then delete the facet object itself, as described in Removing a Facet.
- If you have more than one frame displaying an embedded part, remember that removing the embedded part from one of your display frames does not automatically remove it from the others. If the removed frame is synchronized with embedded frames in your other display frames, for example, you must remove those embedded frames also.
- Do not create a link specification for the data you have cut; the data no longer belongs to your part.
- If the data you cut to the data-transfer object includes a link source, you must call the SetSourcePart method of the link source object in your own draft, passing it a null value. That action relinquishes your part's ownership of the link source (except that you retain a reference to it in your undo data). Even if you write a promise instead of actual data, call SetSourcePart at this time; don't wait until the promise must be fulfilled.
- If you write a promise to the clipboard when cutting data from your part, you can later fulfill that promise using either a kODCloneCut or kODCloneCopy transaction (unless the storage-unit view contains a kODPropCloneKindUsed property; see Fulfilling a Promise). You have this flexibility because the same promise may need to be fulfilled during a paste immediately following the cut, or to subsequent pastes that do not follow the cut. Therefore, you must take all cut-specific actions at the time you write the promise (or, if the cut is undone, when you handle the undo action.)
- If you have cut data from your part and cloned it to the clipboard or drag and drop object, OpenDoc may use the actual objects that were cut from your part-not clones of them-when providing the data to be pasted or dropped into a destination. Therefore, it is

important to release, rather than delete, objects (embedded frames, links, and so on) that you cut from your part. Likewise, if you paste data into your part and then the user selects Undo, make sure that you release rather than delete the objects that you remove in the course of reversing the paste operation.

Handling Pasted or Dropped Data

When data is pasted or dropped into a part, the part receiving the data can either embed the data as a separate or incorporate the data as intrinsic content. The part may also, in some circumstances, translate the data, or it might even refuse to accept the pasted data.

This section discusses how OpenDoc and your part editor make these decisions, both with and without explicit user intervention. It also discusses when your part editor might explicitly translate data.

Default Conventions

In the absence of other instructions, OpenDoc expects your part to follow these specific conventions when pasting data from the clipboard or when accepting dropped data during a drag and drop operation.

- If the transferred data consists of an arbitrary portion of the source part's intrinsic content-plus possibly one or more embedded parts-the destination part either incorporates or embeds that outer intrinsic content, according to these rules:
    - If any of the representations of the outer data are of a part kind directly readable by the destination part editor, the editor should incorporate the outer data into the intrinsic content of the destination part, and embed any parts that were embedded in the outer data.
    - If none of the representations of the outer data is readable by the destination part editor, the editor should transfer the data as a single part (plus any embedded parts), and embed the new part in a frame in the destination.
- If the transferred data represents a single embedded frame that was cut or copied (or dragged), the destination part editor should embed the data as a separate part into the destination part, regardless of whether its part kind is different from or identical to that of the destination.

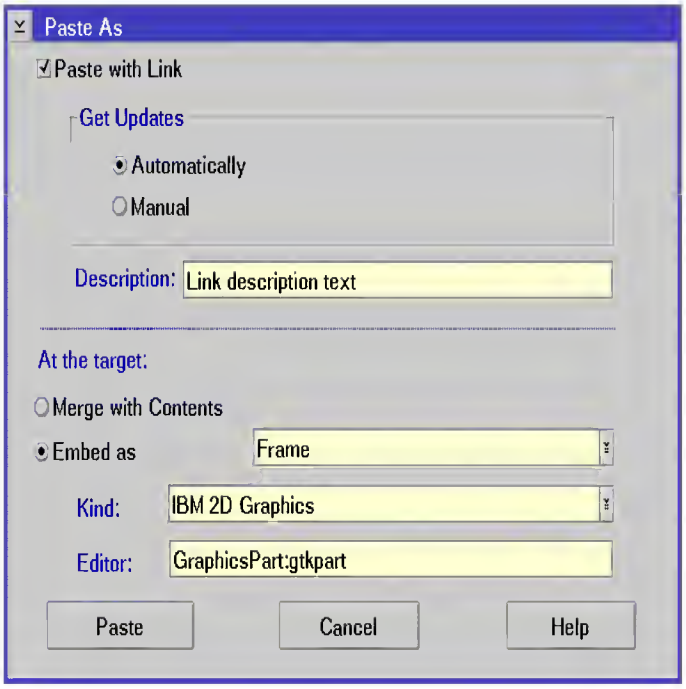
The destination part should place clipboard data at the insertion point; it should place dropped data at the point where the user releases the mouse button. Note also that a pasted or dropped part takes on the view type that its containing part prefers it to have.

Handling the Paste As Dialog Box

The Paste As dialog box allows the user to override OpenDoc and specify whether transferred data is to be embedded, incorporated, or translated and then embedded or incorporated. It also allows the user to create a link to the source of the transferred data.

When the user chooses the Paste as command in the Edit menu (see Edit Menu), the active part calls the clipboard's ShowPasteAsDialog method to display the Paste As dialog box, shown in the following figure. Also, when the user presses Alt in a drag and drop operation, the active part calls the ODDragAndDrop ShowPasteAsDialog method.

A part at the destination of a drop also displays this dialog box if the user holds down the Command key while performing a drop; see Dropping.



The user can select the following options from the dialog box:

- **Paste with link.** The user can request that a link be created between the source and destination data. This option is not available if no link specification accompanies the data to be pasted. Disable this option if your part does not support linking.
- **Get Updates (of a link) Automatically/Manually.** If the user creates a link, the user selects one of these options to specify whether updates are to be automatic or manual. See Automatic and Manual Propagation of Updates for more information.
- **Merge with contents.** If the user chooses this option, the destination part editor is expected to incorporate the data, if at all possible, even if doing so requires translation and great loss of information (such as converting text to a bitmap).

  When your part editor displays the dialog box, it can check the part kinds available in the data-transfer object and specify whether this option is enabled and whether it is checked by default. Disable this option if your part cannot incorporate the data, even after translation.
- **Embed As.** If the user chooses this option, the source data must be embedded in the destination as a separate part, even if incorporation is possible.

  When your part editor displays the dialog box, it can check the part kinds available in the data-transfer object and specify whether this option is enabled and whether it is checked by default. Disable this option if your part does not support embedding.

  If the user selects Embed As, the user can also make the following selections:

  - **Kind (of pasted data).** With this option, the user can override OpenDoc's default pasting decision and explicitly specify a destination part kind from a pop-up menu. The user specifies the specific translation to perform by selecting a new part kind for the part in the kind pop-up menu. The kind pop-up menu allows the user to select a part kind to translate to.

    This option (choosing part kind) is not available if the transferred data consists of a single frame that is being moved (not copied) from its original source. The moved frame may not be the only frame displaying its part, and changing its kind would then affect the display in the other parts, which may not be what the user intended.
  - **Editor.** From the Editor pop-up menu, the user can specify a new editor for the part. Only part editors that can read the currently selected part appear in the menu.
  - **View type.** From a pop-up menu, the user can choose the view type the embedded part is to have: frame, large icon, small icon, or thumbnail.

The ShowPasteAsDialog method returns the results of the user's choices to your part in an ODPasteAsResult structure:

```
struct ODPasteAsResult
{
        ODBoolean           pasteLinkSetting;
        ODBoolean           autoUpdateSetting;
        ODBoolean           mergeSetting;
        ODTypeToken         selectedView;
        ODType              selectedKind;
        ODType              translateKind;
```

```
        ODEditor            editor;
};
```

Depending on the contents of the returned structure, your part either embeds or incorporates the transferred data, either accepts it as it is or translates it, and either creates a link to its source or does not.

- If the user has chosen to create a link, respond as described in Creating a Link at the Destination. If you create a link, take into account the automatic/manual update setting, as well as the other Paste As settings chosen by the user.
- If the user has chosen to translate the data but is not creating a link, respond as described in Translating Before Incorporating or Embedding. If the user is not creating a link or translating, read the transferred data in either of two ways:
    - If your part can directly read any of the part kinds in the data-transfer object's contents storage unit, and if the user has not selected "Embed As", you incorporate the data. Follow the instructions in Incorporating Intrinsic Content.
    - If your part cannot directly read any of the part kinds in the storage unit, or if the user has selected "Embed As", follow the instructions in Embedding a Single Part.

For more on translation, see Binding with Translation.

Writing to a Data-Transfer Object

You write data to a data-transfer object when the user cuts, copies, or drags data from your part or when you create or update the source of a link. This section discusses how to place that data into the data-transfer object.

This section does not discuss how your part handles cutting (or moving) differently from copying. Cutting involves removing data from your part, including possibly one or more frames of embedded parts. Removing an embedded part is discussed in Removing an Embedded Part; other issues related to cutting operations are discussed in Handling Cut Data.

Writing Intrinsic Content

Intrinsic content plus possibly one or more embedded parts is the most general configuration of data that you put into a data-transfer object. If the data to be written consists of a combination of your part's intrinsic content plus embedded parts, you need to write your own intrinsic content, and you need to clone the embedded frames as well. If the data includes link sources or destinations, you need to clone those objects also. These are the basic steps to take, regardless of whether or not your intrinsic content is accompanied by embedded frames and other objects:

1. Gain access to the data-transfer object and prepare to write to it. (See, for example, the initial steps under Copying or Cutting to the Clipboard, Initiating a Drag, Creating a Link at the Source, and Updating a Link at the Source.)
2. Start the cloning operation, as described under Cloning. Specify the appropriate kind of cloning operation, using one of the constants listed in the table shown in Cloning Sequence.
3. Clone the embedded part into the data-transfer object by calling your draft's Clone method. Unlike writing intrinsic content, you do not add a kODPropContents property (the embedded part itself does that), a kODPropName property (OpenDoc does that), or a kODPropSuggestedFrame property (you instead add a kODPropContentFrame property).

   Be sure to perform the cloning operation in this order:

   a. Add a property of type kODPropContentFrame to the data-transfer object's content storage unit. The presence of this property tells a destination part that the data being transferred is a frame without surrounding intrinsic content, and also signals to the embedded part (the part being cloned) that it can write a promise instead of its actual content.

      It is important *not* to clone the frame yet. Wait until step 4, or else the embedded part itself will be cloned into the wrong storage unit.

   b. Clone the embedded part into the data-transfer object's content storage unit.

   c. Clone the embedded part's frame into the data-transfer object's draft (into any storage unit other than the content storage unit). This cloning operation must occur *after* the embedded part is cloned.

   d. Add a value of type kODWeakStorageUnitRef to the kODPropContentFrame property of the data-transfer object's content storage unit. Create a weak persistent reference from that value to the cloned frame. This allows a destination part, upon recognizing the kODPropContentFrame property, to locate the frame for the part in the data-transfer object.

4. Optionally, write any intrinsic data you want associated with the frame (such as a drop shadow or other visual adornment) as proxy content. Add a property (of type kODPropProxyContents) to the data-transfer object, and write your data into it as a value that you recognize. If the transferred part is subsequently pasted into a part that also recognizes that value and knows how to interpret it, the added frame characteristics can be duplicated.

5. If the embedded part is the entire source or destination of a link, you need to write additional proxy content, as described in Writing Linked Content to Storage. follow the instructions in the note Writing Links for Data Transfer.

6. If appropriate, write a link specification into the data-transfer object, as described in Link Specification.

7. Perform any closing tasks specific to the individual kind of data-transfer object you are writing to. (See, for example, the final steps under Copying or Cutting to the Clipboard, Initiating a Drag, Creating a Link at the Source, and Updating a Link at the Source.)

If this operation is a cut rather than a copy, note the additional considerations listed in Handling Cut Data.

Reading from a Data-Transfer Object

You read from a data-transfer object when the user pastes or drops data into your part or when you create or update the destination of a link. This section discusses how to extract that data from the data-transfer object.

When placing transferred data into your part, what you do with the data depends on how the part kinds within the transferred data relate to the part kind of your part (the destination part).

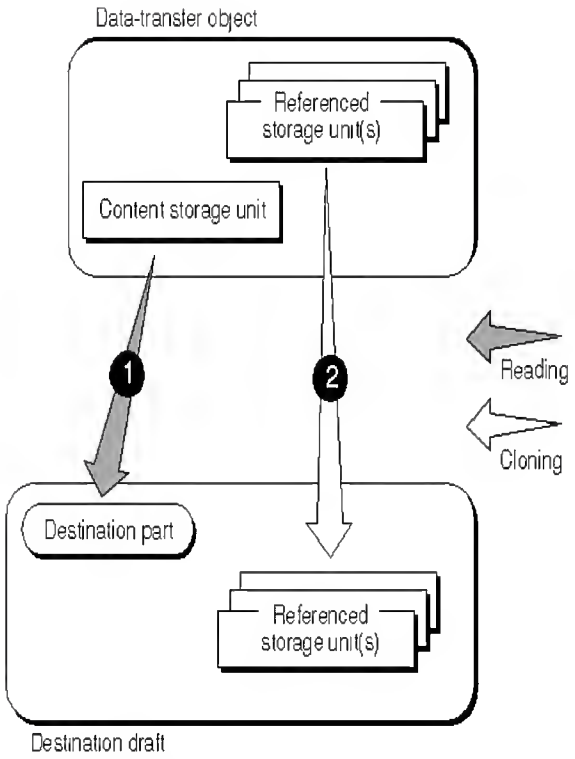
Incorporating Intrinsic Content

As the destination of a data transfer, your part incorporates the intrinsic content of the data-transfer object into your own part's intrinsic content (and embeds whatever embedded parts the transferred content contains) if all of these conditions apply:

- The intrinsic content of the transferred data is stored in at least one part kind that you can incorporate into your part.
- The user has not specified Embed As in the Paste As dialog box.
- The data-transfer object's content storage unit does not contain a property named kODPropContentFrame. (If it does, the transferred data consists of a single embedded frame without surrounding intrinsic content, and the data should be embedded; see Frame Shape or Frame Annotation.)

When incorporating transferred data, you should read the highest-fidelity part kind possible-that is, the first value (in storage order) that your part editor understands. If the transferred data includes a kODPropPreferredKind property, however, it takes precedence over fidelity; you should attempt to read it first.

Incorporating involves reading intrinsic content plus possibly cloning embedded frames, links, and other objects. The following figure summarizes the steps involved.



1. Read content into your part
2. Clone referenced storage units into your draft

Here, in more detail, are the basic steps to take when incorporating:

1. Gain access to the data-transfer object and prepare to read from it. (See, for example, the initial steps under Pasting from the Clipboard, Dropping, Creating a Link at the Destination, and Updating a Link at the Destination.) If you are incorporating translated data, you already will have cloned the data into a temporary storage unit in your draft, and you already will have translated it. Take these steps:

    - In the kODPropContents property of that storage unit, focus on the value that corresponds to the translated part kind. Read the data into your part, following your own content model.
    - Skip to step 5.

2. Start the cloning operation, as described under Cloning. Specify the appropriate kind of cloning operation, using one of the constants listed in the table shown in Cloning Sequence.

3. In the kODPropContents property of the data-transfer object's content storage unit, focus on the value that corresponds to the highest-fidelity part kind that you can incorporate into your part. (It may not be the highest-fidelity value present in the property.) Read the data into your part, following your own content model.

    As you encounter persistent references to objects-embedded frames, link-source objects, link objects, auxiliary storage units, and so on-clone each object into your draft, by calling the Clone method of the data-transfer object's draft (see "Clone" under Cloning Sequence.) Adjust your persistent references to point to the newly cloned objects. (Remember to retain the cloned objects IDs for instantiation after cloning is complete, rather than reconstructing the objects yet.)

4. End the cloning operation, as described in "EndClone" in Cloning Sequence.

5. If you are incorporating as a result of a drop, there may be a kODPropMouseDownOffset property in the data-transfer object's content storage unit. If so, focus on that property, read its value, and-if appropriate for your content model-use the value to position the incorporated data in relation to the drop location.

    (If you are incorporating translated data, you must read this property from the original data-transfer object, not the cloned storage unit.)

6. If the cloning was successful, instantiate each cloned embedded frame into memory (with your draft's AcquireFrame method) and call the frame's SetContainingFrame method, to make your part's display frame (the frame that received the paste) the containing frame of the new embedded frame.

    Create additional embedded frames as necessary, if your part's content model specifies that the new part is to appear in more than one of your display frames. If you do create new frames, synchronize them with the first (source) frame; see Synchronizing Display Frames for more information.

7. Change each new frame's link status to reflect its current location. If your part does not support linking, you must nevertheless change your cloned embedded frames' link status (to kODNotInLink).

8. If any of the objects that you have cloned into your part is a link-source object or link object, follow the procedures described in Reading Linked Content from Storage to make sure that the objects are valid.

9. If any newly embedded frame is visible, assign a facet or facets to it, as described in Adding a Facet.

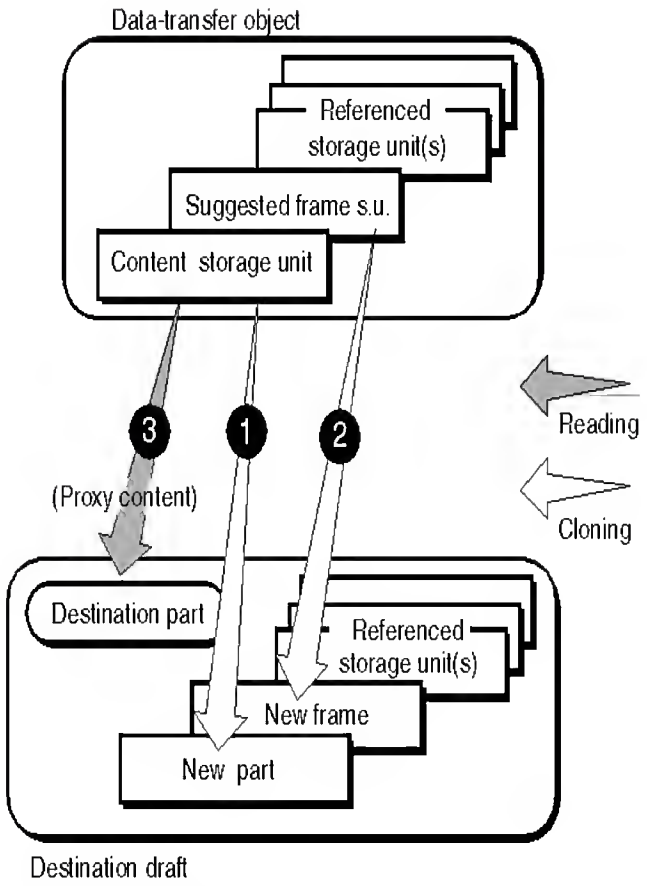
10. Perform any closing tasks specific to the individual kind of data-transfer object you are reading from. (See, for example, the final steps under Pasting from the Clipboard, Dropping, Creating a Link at the Destination, and Updating a Link at the Destination.) (If you are incorporating translated data, you can at this time remove the cloned temporary storage unit from your draft.)

Embedding a Single Part

As the destination of a data transfer, your part embeds the entire contents of the data-transfer object as a single part (plus whatever embedded parts it contains) if any of these conditions apply:

- The intrinsic content is of a part kind that you cannot incorporate.
- The user has specified Embed As in the Past As dialog box.
- The transferred data consists of a single embedded frame, without surrounding intrinsic content (regardless of its part kind). In this case the data-transfer object's content storage unit contains a property named kODPropContentFrame, that is a signal that the data consists of a single frame and should be embedded, even if it is of a part kind that you can incorporate. See Frame Shape or Frame Annotation.

Embedding data from a data-transfer object as a single part involves, basically, cloning the content storage unit into your draft and then providing for a frame and facets for the new part. The steps involved are summarized in the following figure.



1. Clone content storage unit into a new part in your draft (referenced storage units are cloned also)
2. Clone suggested frame (if present) into your draft
3. Read proxy content (if present) into your part

Here, in more detail, are the basic steps to take when embedding:

1. Gain access to the data-transfer object and prepare to read from it. (See, for example, the initial steps under Pasting from the Clipboard, Dropping, Creating a Link at the Destination, and Updating a Link at the Destination.)

   If you are embedding translated data, you already will have cloned the data into a new storage unit in your draft, and you already will have translated it. Skip to step 6.

2. Start the cloning operation, as described under Cloning. Specify the appropriate kind of cloning operation, using one of the constants listed in the table shown in Cloning Sequence.

3. Clone the data-transfer object's content storage unit into a new storage unit in your draft, using the Clone method of the data-transfer object's draft.

4. If there is a property named kODPropContentFrame in the original storage unit, read the storage-unit reference it contains and use that reference to clone the new part's frame into your draft. (Cloning the data-transfer object's content storage unit alone does not copy the frame, because the reference is a weak persistent reference.)

5. End the cloning operation. If any of the following conditions apply, notify the embedded part's future part editor that it should use a specific part kind when reading the part:

   - If the data has been translated.
   - If this embedding has occurred as a result of a user selection in the Paste As dialog box and the user has chosen a part kind that is not the preferred kind.
   - If a preferred kind property does not exist and the user has chosen a part kind that is not the highest-fidelity (first) value stored in the transferred storage unit's contents property.

   In any of these cases, create a property with the name kODPropPreferredKind in the cloned storage unit (if the property does not already exist) and write into it a value that specifies the part kind the editor should use.

6. If this embedding has occurred as a result of a user selection in the Paste As dialog box and the user has chosen a specific part

editor to edit the part, add a property with the name kODPropPreferredEditor to the cloned storage unit, and write into it the editor ID (returned in the editor field of the ODPasteAsResult structure) of the preferred editor.

7. If the original storage unit contains a property named kODPropProxyContent, that property contains any proxy content that the part's original containing part wanted associated with the frame, such as a drop shadow or other visual adornment. (This property is absent if the transferred data includes any intrinsic content in addition to the embedded frame.)

   Focus the cloned storage unit on the kODPropProxyContent property and read in the information from the data-transfer object (not from the cloned storage unit). You must understand the format of the proxy content in order to use it; it is subsumed in your own part's intrinsic content to be associated with the frame. If you do not understand the format, ignore the data.

8. If you are embedding as a result of a drop, there may be a property named kODPropMouseDownOffset in the content storage unit of the data-transfer object (not the cloned storage unit). If so, focus on that property, read its value, and-if appropriate for your content model-use the value to position the embedded data in relation to the drop location.

9. Recreate the new part's frame-if it has been provided in the kODPropContentFrame property-using your draft's AcquireFrame method, and call its SetContainingFrame method to assign your part's display frame (the frame that received the paste) as the containing frame. If no frame was provided, you need to create one:

   - Obtain the suggested frame shape-if it exists-from the data-transfer object (not from the cloned storage unit). It is in a property named kODPropSuggestedFrameShape. If it is not there, use a default frame shape.
   - Recreate the new part; pass the cloned storage unit's ID to your draft's AcquirePart method.
   - Create the frame for the part, using your draft's CreateFrame method.

10. Change the link status of the new frame to reflect its current location. If your part does not support linking, you must nevertheless change your new embedded frame's link status (to kODNotInLink).

11. If the newly embedded frame is visible, assign facets to it, as described in Adding a Facet.

12. Perform any closing tasks specific to the individual kind of data-transfer object you are reading from. (See, for example, the final steps under Pasting from the Clipboard, Dropping, Creating a Link at the Destination, and Updating a Link at the Destination.)

Translating Before Incorporating or Embedding

If you must translate transferred data before incorporating or embedding it in your part, follow the steps listed here before following those in Incorporating Intrinsic Content and Embedding a Single Part. The OpenDoc translation service is described in general in Translation. Basically, you translate by first cloning the data into a storage unit in your own draft, then modifying it, and then completing the data transfer. When you translate transferred data, always write the translated data into a storage unit in your own draft; do not write it back into the original data-transfer object. Subsequent readers of the data cannot tell a translated value from a value written by the data's original part editor.

These are the general steps you might follow:

1. Gain access to the data-transfer object that contains the untranslated data and prepare to read from it. (See, for example, the initial steps under Pasting from the Clipboard, Dropping, and Updating a Link at the Source.

   (If you are creating a link, remember that you read the data from the newly created link object, not the clipboard or drag and drop that contained the link specification.)

2. Start the cloning operation, as described under Cloning. Specify the appropriate kind of cloning transaction, using one of the constants listed in the table in Cloning Sequence.

3. Clone the data-transfer object's content storage unit into a new storage unit in your draft, using the Clone method of the data-transfer object's draft.

4. If there is a property named kODPropContentFrame in the original storage unit, you must embed the data (unless the user explicitly specified incorporation in the Paste As dialog box). Read the storage-unit reference the property contains and use that reference to clone the new part's frame from the data-transfer object's storage unit into your draft. (Cloning the data-transfer object's content storage unit alone does not copy the frame, because the reference is a weak persistent reference.)

5. Access the storage unit of the cloned-but-untranslated data, and focus on its contents property. Add a value whose value type is the part kind you are translating to.

6. Create two storage-unit views: one focused on the original untranslated value, and the other focused on the translated value you have just created.

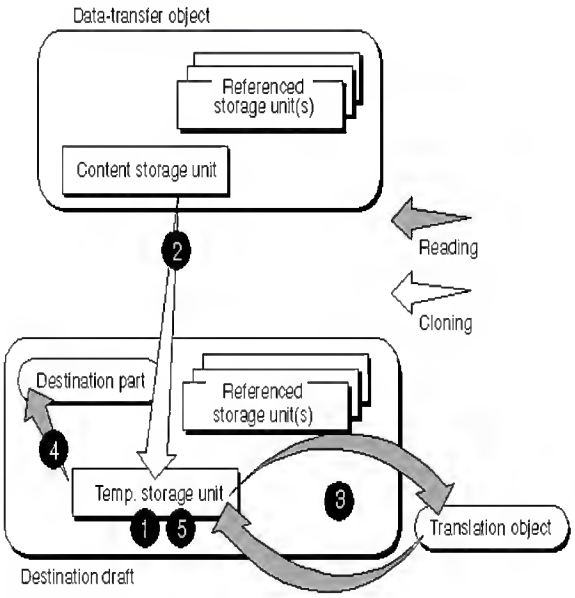
7. Gain access to the translation object (by calling the session object's GetTranslation method), and then call the translation object's TranslateView method, passing it the two storage-unit views. The translation object performs the translation and writes the new data into the new value.

8. Delete the storage-unit views.

Complete the transfer of the translated data in either of two ways, depending on whether you are incorporating or embedding, or whether you are creating or updating a link:

- If your part can directly read the part kind of the translated data (and if the user has not selected "Embed As" from the Paste As dialog box), incorporate it. Follow the steps in the section Incorporating Intrinsic Content.

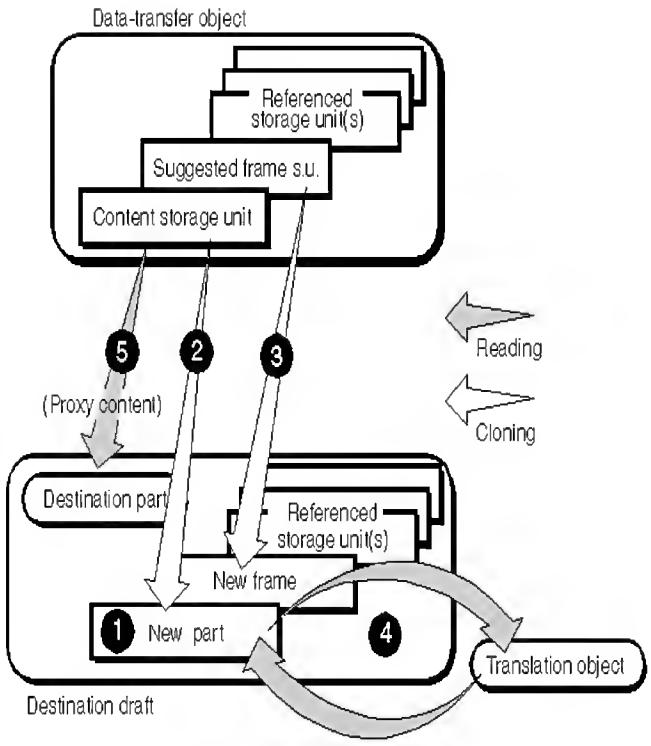
  The following figure summarizes the steps involved. Note that you read the content into your part from the cloned temporary storage unit, not the original data-transfer object, and that you remove the temporary storage from your draft when the transfer is complete.



1. Create a temporary storage unit in your draft
2. Clone content storage unit into temporary storage unit (referenced storage units are cloned also)
3. Translate content of temporary storage unit
4. Read translated content into your part
5. Remove temporary storage unit from your draft

- If your part cannot directly read the part kind of the translated data (or if the user has selected "Embed As" from the Paste As dialog box), follow the instructions in Embedding a Single Part.

  The following figure summarizes the steps involved in this case. Note that the cloned storage unit is not temporary, but becomes the new embedded part's storage unit. Note also that you must read proxy content from the original data-transfer storage unit, not the cloned one.



1. Create a storage unit for the new part
2. Clone content storage unit into the new part (referenced storage units are cloned also)
3. Clone suggested frame storage unit (if present)
4. Translate content of new part
5. Read proxy content (if present) from data-transfer object

Clipboard Transfer

The clipboard commands Cut, Copy, and Paste constitute a common mechanism for transferring data within and across conventional documents, even documents of different data formats. For OpenDoc documents, these commands perform the same tasks, but with added capabilities:

- The commands operate on embedded parts as well as intrinsic content. The data copied to or pasted from the clipboard can contain any number of parts, of any part kinds. The parts may be displayed in frames or represented as icons.
- The **Paste** and **Paste As** commands can either embed parts or incorporate data as intrinsic content.
- The **Paste** and **Paste As** commands can cause a portion of one part's intrinsic content to become, when copied and then pasted into another part, a separate, embedded part on its own.
- The **Paste As** command can create a persistent link to the source of the data being pasted.
- The **Paste As** command also allows the user to override decisions on incorporating vs. embedding that would normally be made by OpenDoc.

As with data placed in a conventional clipboard, if your part editor stores multiple formats on the clipboard-including standard formats-the user has a greater chance of being able to incorporate (rather than embed) the data into more kinds of parts.

Clipboard Concepts

This section discusses some basic clipboard operations common to both reading and writing, such as acquiring the clipboard focus and using

the clipboard update ID. It also presents some special considerations related to updating the clipboard and cutting data from your part.

Enabling the Cut and Paste menu items
OpenDoc does not permit changes to your draft (and therefore your part's content) if the draft permissions are read-only. You should check your draft permissions (see Drafts) before enabling the Cut, Paste, or Paste As items in the Edit menu and before allowing the user to perform a cut or paste operation using keyboard equivalents.

The OS/2 implementation of the clipboard object provides two extra methods to help parts in determining whether or not they can enable the Paste menu items; they are called CanEmbed and CanIncorporate. CanEmbed queries the registration manager for a part editor capable of handling the content of the clipboard storage unit. It is intended for use by container parts. CanIncorporate determines if a specific type of content is present on the clipboard. Parts should call this method for the kinds they support.

Acquiring and Relinquishing the Clipboard Focus

Your part can access the clipboard only when your part is in the foreground process; access from a background process is meaningless. Furthermore, to be thread-safe, you must always acquire the clipboard focus before writing or reading clipboard data. As long as you hold the clipboard focus, no other part can access or modify the data.

Typically, your part needs to inspect the contents of the clipboard before deciding whether to enable Edit menu items, so your AdjustMenus method can include a call to the arbitrator's RequestFocus method to acquire the clipboard focus.

After acquiring the clipboard focus, you can access the clipboard when responding to a menu command (or its keyboard equivalent) with your HandleEvent method by calling the session object's GetClipboard method. You should then unilaterally relinquish the clipboard focus in your HandleEvent method, after having handled the clipboard command. (Your HandleEvent method is always called after AdjustMenus is called, even if the user does not choose a menu command.)

These are the steps you take to acquire the clipboard focus and prepare to write to it or read from it:

1. Acquire the clipboard focus, using the Arbitrator's RequestFocus method (as described in Requesting Foci).
2. Get access to the clipboard object, using the session object's GetClipboard method.
3. Acquire the clipboard focus, using the Arbitrator's RequestFocus method (as described in Requesting Foci).
4. If you are writing to the clipboard, remove any existing data on the clipboard with the clipboard's Clear method. (Do not take this step if you are reading from the clipboard.)
5. Get access to the clipboard's content storage unit, using the clipboard's GetContentStorageUnit method.

You relinquish the Clipboard focus by calling the arbitrator's RelinquishFocus method.

Clipboard Update ID

Whenever you copy data to the clipboard, you should get and save the clipboard's current *update ID* , a number used to identify this particular instance of clipboard contents. You typically obtain the update ID in this situation by calling the clipboard's ActionDone method. In other situations, you may need to inspect the clipboard's current update ID; you can do so by calling the clipboard's GetUpdateID method.

If a link specification that you have written to the clipboard becomes invalid because of changes to your content that was copied to the clipboard, you must remove the link specification from the clipboard, as described in Removing a Link Specification from the Clipboard. You can examine the clipboard's current update ID at any time and compare it with your saved update ID; if they are identical, the clipboard has not changed.

Removing a Link Specification from the Clipboard

If your part copies some of its content to the clipboard and the user then modifies the equivalent content in your part (without copying anything else to the clipboard), the clipboard data no longer exactly matches the source data in your part. The potential link represented by the link specification you wrote in the clipboard therefore no longer reflects the content at the source of the link. If your source content has changed to the extent that creating a link is no longer feasible, your part must remove the link specification.

By saving the update ID of the clipboard whenever you copy data to it, you can check that ID against the current update ID of the clipboard

whenever your source data changes to the extent that creating a link to the previous data is impossible. If the IDs match, the clipboard still contains the data that you placed in it, and you should remove the link specification.

You can follow these steps to remove a link specification:

1. Acquire the clipboard focus as described in Acquiring and Relinquishing the Clipboard Focus.
2. Get the clipboard's update ID and compare it to your stored update ID. If they do not match, skip to step 5.
3. Access the clipboard's content storage unit. (Unlike when writing to the clipboard, do not clear the clipboard data.)
4. Focus the storage unit on the link-specification property (type kODLinkSpec), and remove that property, using the storage unit's Remove method. Be sure to focus on the entire property (by specifying a null value type), so that you remove the entire property, not just one value.
5. Call the clipboard object's ExportClipboard method (OS/2 implementation only).
6. Relinquish the clipboard focus.

Undo for Clipboard

If your part supports cutting, copying, or pasting, it must also support undoing those operations. (Note that data transfer between parts is undoable only if both parts involved have undo support.) Undo support in general is described in Undo.

Whenever your part performs a cut, copy, or paste operation, it must call the clipboard's ActionDone method, to notify the clipboard of the kind of cloning transaction that took place. ActionDone returns a clipboard ID that identifies the current clipboard contents. Save that update ID.

When your part cuts an object to the clipboard, it should remove the object from its content data structures. However, your part should not release the object, but instead save a reference to the object in an undo action. (For a cut embedded frame, you also set its in-limbo flag to true, as shown in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames. Then, if the user chooses the Undo command and your part's UndoAction method is called, your part should:

- Reinstate the object in your part's content structures (and setting its in-limbo flag to false).
- Notify the clipboard that the cut was undone, by calling the clipboard's ActionUndone method, passing it the update ID returned to you when you called ActionDone after cutting the data to the clipboard. If the user subsequently chooses the Redo command and your part's RedoAction method is called, your part should
- Once again remove the object from its content data structures (and reset its in-limbo flag to true)
- Notify the clipboard that the cut was redone, by calling the clipboard's ActionRedone method, passing it the same update ID

Calling ActionDone, ActionUndone, and ActionRedone notifies the clipboard whether it is involved in a cut, a copy, or the restoration of a cut or a copy; the clipboard's internal handling of its objects differs in each case.

If and when your part's DisposeActionState method is called, you can at that point release (not remove from your draft) the object referenced in your undo action. (For an embedded frame, you either release or remove it, as shown in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames.)

To undo a cut, copy, or paste operation requires the restoration of the previous state of the document involved, but it does not require the restoration of the previous state of the clipboard. Therefore, if you redo a paste operation that has been undone, you cannot assume that the clipboard once again contains the original data that had been pasted. You must implement the redo from your own data. For this reason, you should retain a copy of pasted data in a private cache.

Copying or Cutting to the Clipboard

You write data to the clipboard as a result of the user selecting the Cut command or the Copy command from the Edit menu (or their keyboard equivalents). These are the basic steps to take:

1. Acquire the clipboard focus and access the clipboard's content storage unit, as described in Acquiring and Relinquishing the Clipboard Focus.
2. Write the data to the clipboard.
    - If the selection consists of a combination of your part's intrinsic content plus zero or more embedded parts, you need to

write your own intrinsic content to the clipboard, and you need to clone the embedded frames and parts as well (or you can write a promise). Follow the steps listed in Writing Intrinsic Content.

- If the selection consists of a single frame of an embedded part, with no surrounding intrinsic content, you need to clone the part and provide a frame for it.

In either case, when you call the BeginClone method, specify either kODCloneCopy or kODCloneCut, depending on whether you are copying or cutting the data to the clipboard.

3. When you have finished, call the clipboard's ActionDone method, specifying either kODCloneCopy or kODCloneCut. Save the clipboard's current update ID (see Clipboard Update ID), which is returned from ActionUndone, in case you later have to undo the cut or copy, remove a link specification, or fulfill a promise you wrote.
4. If this operation was a cut rather than a copy, it must be undoable. Add a single action to the action history, as described in Adding an Action to the Undo History. Be sure to follow the special instructions in Handling Cut Data, including setting the in-limbo flag of any cut frame to true.
5. Delete the cut selection from your part's content.
6. Call the clipboard object's ExportClipboard method (OS/2 implementation only).
7. Relinquish the clipboard focus (see Acquiring and Relinquishing the Clipboard Focus).

Pasting from the Clipboard

You read data from the clipboard as a result of the user selecting the Paste command or the Paste As command from the Edit menu. These are the basic steps to take:

1. Acquire the clipboard focus and access the clipboard's content storage unit, as described in Acquiring and Relinquishing the Clipboard Focus. (Do not clear the clipboard data, of course).
2. Read the data from the clipboard.

    - If the data consists of intrinsic content plus zero or more embedded frames, and if the intrinsic content is of a part kind that you can incorporate into your part, you need to read the intrinsic content and possibly clone embedded parts, links, or other objects. Follow the steps listed in Incorporating Intrinsic Content.
    - If you need to embed the data as a single part with no surrounding intrinsic content, you need to clone the part and either extract its frame or create a frame for it. Follow the steps listed in Embedding a Single Part. That section lists the conditions under which you must embed rather than incorporate clipboard data.

    In either case, when you call the BeginClone method, specify kODClonePaste.

3. Pasting must be an undoable operation. Call the clipboard's ActionDone method, specifying kODClonePaste. Save the current clipboard update ID, returned by ActionDone, in a single undo action (see Adding an Action to the Undo History, so that you can undo the paste operation if necessary. If you are pasting an embedded frame, follow the instructions listed in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames. Save the current value of the frame's in-limbo flag and then set the flag to false.
4. Notify OpenDoc and your containing part that there has been a change to your part's content; see Making Content Changes Known.
5. Relinquish the clipboard focus (see Acquiring and Relinquishing the Clipboard Focus).

Drag and Drop

The OpenDoc drag and drop facility allows users to apply direct manipulation to move or copy data. Users can drag items from one location in a part or document (or the desktop) to another location in the same or different part or document (or to the desktop).

Drag and Drop Concepts

Drag and drop provides a direct-manipulation alternative to the clipboard. The fundamental user-level operations-copying and moving-are similar in drag and drop and in clipboard transfer, and are similar across all platforms.

User Interaction

The user typically initiates a drag by positioning the mouse pointer over some selected (or single-click-selectable) content, pressing and holding down the mouse button, and then moving the pointer. The user can hold down a modifier key while pressing the mouse button or while releasing the mouse button to force a drag to be a copy (called a *drag-copy*) or a move (called a *drag-move*).

As the user moves the mouse pointer, an outline of the selected item (provided by the source part, drawn by OpenDoc) is dragged to the new location. When the user releases the mouse button, the item is placed (dropped) at the pointer location. The part beneath the pointer is notified that something has been dropped on it. At this point, the destination part behaves essentially as it does when pasting from the clipboard; it makes the same decisions regarding embedding versus incorporating data and handling links. The source part may have to delete the items if it was a move operation or do nothing if it was a copy.

Parts can restrict the dropped content they will accept to specific part kinds. To indicate that it can accept a drop, a part provides appropriate feedback to the user once the mouse pointer enters its facet.

For frames and icons that are being dragged, OpenDoc provides specific behavior and appearance for the item being dragged. For intrinsic content that is being dragged, your part editor is responsible for defining behavior and providing user feedback.

The user can move an active frame by dragging its border. By moving the pointer to the active frame border and pressing the mouse button, the user selects the frame and can start dragging immediately.

The user can move a selected frame by moving the pointer anywhere within it and pressing the button to start a drag. Also, if a frame is bundled, the user can drag it by pressing the mouse button while the pointer is anywhere in the interior of the frame, whether or not the frame is already active or selected.

When the user releases the mouse button, the frame and its part are dropped at the new location. After a frame is dropped, it becomes selected. The destination may adjust the drop location of a frame based on constraints such as gridding (in a graphics part) or text position (in a text part).

Move Versus Copy

OpenDoc follows these conventions for drag and drop:

- When a user drags an item within a document, the item is moved, not copied. Note that a window boundary is not always a document boundary; dragging an item to or from a part window, but still within the same document, results in a move.
- When the user drags an item across a document boundary, the item is copied, not moved.
- When the user employs drag and drop to move a frame, the moved frame displays the original part if the destination is in the same document. However, if the destination is in a different document, the source frame is deleted and a new part (and new frame displaying the contents) is created at the destination. (In contrast, clipboard pasting always creates a new part and frame.)

OpenDoc supports a mechanism for you to allow the user to force a copy or move operation by pressing modifier keys.

Dragging stationery
Stationery parts may be copied exactly like any other part. However, if you drag-move stationery into a part whose preferred view type for embedded parts is frame view, a copy is "torn off" the stationery pad and displayed in a frame. The stationery part remains in its original location, unchanged.

Droppable Frames

Each frame has a SetDroppable method, through which the part displayed in the frame controls whether or not the frame can (in general) accept dropped data. When a display frame is added or reconnected to your part, you can call SetDroppable and pass either kODTrue or kODFalse to allow or prohibit dropping. (When initially created, frames are not droppable.) You can call SetDroppable again at any time to change the state of the frame.

OpenDoc calls the IsDroppable method of the frame before making any of the drag-related method calls DragEnter, DragWithin, DragLeave, or Drop to your part. If your display frame is not droppable, your part does not receive any of these calls.

Your part's display frame does *not* have to be droppable for you to initiate a drag from it.

Undo for Drag and Drop

If your part supports dragging and dropping data, it must also support undoing that operation. (Note that data transfer between parts is undoable only if both parts involved have undo support.) Undo support in general is described in Undo.

For drag and drop, undo involves a multistage action. The part initiating the drag begins the action, the part receiving the drop adds a single action, and the initiating part completes the action when the drop completes. See Adding Multistage Actions for more details. If drag and drop involves embedded frames, be sure to set the dragged frames' in-limbo flags appropriately when initiating a drag, when dropping, and when the drag completes. See the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames for details.

Initiating a Drag

In response to a mouse-down event at the appropriate location, the part receiving the mouse event (the source part) has the choice of initiating a drag.

You should follow the procedures listed under Mouse Events, Activation, and Dragging in deciding whether to initiate a drag operation.

To initiate a drag, you (the source part) should follow these steps:

1. Call the GetDragAndDrop method of the session object to get access to the ODDragAndDrop object. Then call the Clear method and the GetContentStorageUnit method of the drag and drop object to get an empty storage unit into which to copy the dragged data.

   Note that there is no drag and drop focus or lock to be acquired. Only one part at a time can initiate and complete a drag.

2. Write the dragged data (or write a promise) into the drag and drop storage unit: If the data is intrinsic content-with or without one or more embedded parts-follow the procedure given in Writing Intrinsic Content.

   When you call the Begin Clone method, specify kODCloneCopy, even if you are initiating a drag-move. (The user can override the move-versus-copy conventions when dropping the data.)

3. Create a property with the name kODPropMouseDownOffset in the drag and drop storage unit, and write into it a value (of type kODPoint) that specifies the offset of the mouse from the origin of the selection or item that is being dragged. This offset allows the destination part to correctly locate the item in relation the mouse-up event when it is dropped.

4. If there are frames that should not accept a drop from this drag-such as any frames that are themselves being dragged-call the frames' SetDragging method and pass a value of kODTrue to notify OpenDoc that it should not allow them to be drop targets.

5. If the drag involves an embedded frame, follow the instructions listed in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames. Set the frames's in-limbo falg to true if a drag-move is possible.

6. Add a beginning action to the undo action history (see Adding Multistage Actions), to allow the user to undo the drag if necessary.

7. Initiate the drag by calling the drag and drop object's StartDrag method. When you call StartDrag, you are responsible for providing OpenDoc an image, such as an outline, for it to display to the user as dragging feedback.

The StartDrag method completes when the drop occurs; see Completion of StartDrag.

Operations while a Drag Is in Progress

As long as the mouse button remains pressed, the drag is in progress. Potential destination parts need to perform certain actions during dragging, when the mouse pointer is within their facets.

On Entering a Part's Facet

Any facet the mouse pointer passes over during a drag represents a potential drop destination, if the facet's frame is droppable. OpenDoc calls a part's DragEnter method when the pointer enters one of its droppable frames' facets during a drag:

```
ODDragResult DragEnter (in ODDragItemIterator dragInfo,
                                    in ODFacet facet,
                                    in ODPoint where);
```

On receiving a call to its DragEnter method, your part (the potential destination part) should take these steps:

1. Examine the part kinds of the dragged data, using a drag-item iterator (class ODDragItemIterator) passed to it. Inspect the part kinds in each item's storage unit and determine whether or not you can accept the dragged data.

   Note: As for the clipboard object,the OS/2 implementation of the ODDragAndDrop object provides two extra methods to help parts in determining whether or not they can accept the dragged data; they are called CanEmbed and CanIncorporate.

   CanEmbed queries the registration manager for a part editor capable of handling the content of the drag item. If a match is found, the corresponding rendering mechanism and format pair will be saved in the drag item storage unit (under the the kODSelectedRMF value of the kODPropContents property) to be used later for rendering in the event a drop ocurrs in this facet. This method should be used by container parts.

   CanIncorporate determines if the drag item content matches a specific kind. Parts should call this method for the kinds they support. As for CanEmbed, if a match is found, the corresponding rendering mechanism and format pair will be saved in the drag item storage unit.

   If your part cannot accept all the dragged items, you should not accept a drop at all.

2. If your part can accept a drop, provide the appropriate feedback to the user, such as adorning its frame border or changing the cursor appearance; Otherwise, take no action.

   When the user initiates a drag from within your frame, you should not display destination feedback as long as the mouse pointer has not yet left your frame, although you should provide feedback if the mouse pointer returns to your frame after having left it.

   You can draw the drag feedback on your own, or with the help of platform-specific services.

3. Examine the current state of the user's system, if desired, to obtain any relevant information (such as whether the user has pressed a modifier key since initiating the drag).

   If you return kODFalse from this method, OpenDoc assumes you cannot accept a drop and will not subsequently call your DragWithin, DragLeave, or Drop methods as long as the mouse pointer remains in this facet.

While within a Part's Facet

OpenDoc calls the potential destination part's DragWithin method continuously while the mouse pointer remains inside the facet:

```
ODDragResult DragWithin(in ODDragItemIterator dragInfo,
                                    in ODFacet facet,
                                    in ODPoint where);
```

In response, the part can do any desired processing inside of its display. For example, if a part allows objects to be dropped only in individual hot spots, it may change its feedback based on mouse-pointer location.

Calls to a part's DragWithin method also give the part an additional chance to examine the state of the user's system. For example, the part may want to find out whether the user has pressed a modifier key during the drag operation.

If you return kODFalse from this method, OpenDoc assumes you no longer wish to accept a drop in this facet. It will not make additional calls to your DragWithin method, and will not call your DragLeave or Drop method, as long as the mouse pointer remains in this facet.

On Leaving a Part's Facet

OpenDoc calls the DragLeave method of a potential destination part when the mouse pointer leaves a droppable facet:

```
ODDragResult DragLeave (in ODFacet facet,
                        in ODPoint where);
```

In response, the part might remove the adornment on its frame or restore the cursor appearance to its original form.

Dropping

If the user releases the mouse button while the pointer is within a facet, OpenDoc calls the Drop method of the facet's part. This is the interface to the Drop method:

```
ODDragResult Drop (in ODDragItemIterator dropInfo,
                   in ODFacet facet,
                   in ODPoint where);
```

The potential destination part then decides whether it can receive the dragged object. If it accepts the data, it either incorporates it or embeds it. This section describes how to design your drop method, how to accept a drop of non-OpenDoc data, and what the source part (the part that initiated the drag) should do after the drop occurs.

Drag Attributes and the Drop Method

The OpenDoc human interface guidelines specify when a drop should be considered a move, and when it should be a copy. The destination part can inspect the drag attributes (by calling the GetDragAttributes method of the drag and drop object) to determine how to handle the drop. Drag attributes are bit flags, and more than one can be set at a time.

These are the possible drag attributes for a drop:

Constant	Description
koDDragIsInSourceFrame	The item being dragged has not yet left the source frame of the drag.
koDDragIsInSourcePart	The item being dragged has not yet left the source part of the drag.
kODDropIsInSourceFrame	The drop is occurring in the source frame of the drag.
kODDropIsInSourcePart	The drop is occurring in the part displayed in the source frame of the drag (though not necessarily in the source frame itself).
kODDropIsMove	This drag and drop operation is a move.
kODDropIsCopy	This drag and drop operation is a copy.
kODDropIsLink	This drag and drop operation is a link (OS/2 implementation only).
kODDropIsPasteAs	The destination part should display the Paste As dialog box (the user has held down the Command key while dropping).

Your Drop method might follow steps similar to these:

1. Use the ODDragItemIterator passed to you to determine whether you can handle the drop, as described in On Entering a Part's Facet. (Alternatively, you could set a "can-drop" flag in DragEnter or DragWithin that you inspect at this point.)
2. Examine the drag attributes to see, for example, whether this is a move or a copy, and whether or not to display the Paste As dialog box.
3. (OS/2 implementation only) Focus the drag item storage unit on the kODDragitem value of the kODPropContents property and create a storage unit view. Then call the ODDragAndDrop object GetDataFromDragManager to render the data. Upon successful completion,this method will place the rendered data in the returned storage unit.

4. If the drag attribute kODDropIsPasteAs is set (that is, if the user held down the Command key when the drop occurred), take these steps before reading from the drag and drop object:

    - Call the ShowPasteAsDialog method of the drag and drop object to display a Paste As dialog box (see the figure in Handling the Paste As Dialog Box).
    - If the ShowPasteAsDialog method returns a result of true, the user has pressed the OK button; use the results of the interaction (passed back as a structure of type ODPasteAsResult) to determine what action to take. Handling the Paste As Dialog Box lists the kinds of pasting that the user can specify.

5. Depending on the nature of the data and the user's instructions, you may either incorporate the data or embed it, you may first translate it, and you may create a link to its source.

    - If you are creating a link, follow the steps in Creating a Link at the Destination. Take into account the automatic/manual update setting, as well as the other Paste As settings chosen by the user.
    - If you are translating but not creating a link, follow the steps in Translating Before Incorporating or Embedding.
    - If you are simply incorporating the data, follow the steps in Incorporating Intrinsic Content.
    - If you are simply embedding the data, follow the steps in Embedding a Single Part.

    In all of these cases, you initially read the data from the drag and drop object, and you specify the following kinds of clone transactions:

    - If a move is specified (the drag attribute kODDropIsMove is set), use a clone transaction of kODCloneDropMove -unless the user has specified that a link be created. Creating a link takes precedence over specifying a drag-move; therefore, use a clone transaction of kODCloneDropCopy in this case.
    - If a copy is specified (the drag attribute kODDropIsCopy is set), use a clone transaction of kODCloneDropCopy.

    Note: The destination part can override a drag that is a move and force it to be a copy. It cannot, however, override a copy and force it to be a move.

6. Add a single action to the action history (see Adding an Action to the Undo History) so that the user can undo the drop. Adding Multistage Actions describes why this action must be a single-stage action.

    If the drop involves an embedded frame, follow the instructions listed in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames. Save the current value of the frame's in-limbo flag and then set the flag to false.

7. If it is not already active, activate the frame in which the drop occurred, and select the dropped content. When it completes, your Drop method should return an appropriate result (of type ODDropResult), such as kODDropCopy, kODDropMove, or kODDropFail. OpenDoc in turn passes that information on to the source part; see Completion of StartDrag.

8. Notify OpenDoc and your containing part that there has been a change to your part's content; see Making Content Changes Known.

When it completes, your Drop method should return an appropriate result (of type ODDropResult), such as kODDropCopy, kODDropMove, or kODDropFail. OpenDoc in turn passes that information on to the source part; see Completion of StartDrag.

Accepting Non-OpenDoc Data

When your part's Drop method is called, it is passed a drag-item iterator (class ODDragItemIterator), so that you can access all drag items in the drag and drop object. If OpenDoc data has been dragged, there is only one drag item in the object, but if the data comes from outside of OpenDoc, there may be more than one item. It is the responsibility of the destination part to iterate through all the drag items to find out whether it can accept the drop.

You can examine the contents property of the storage unit of each dragged item to determine the kind of data it consists of. If you can read data of that type, you can incorporate it into your part. Otherwise, you may be able to embed it as a separate part.

- To access a file whose data you intend to incorporate, you can obtain the item's HFSFlavor structure from a a value in the contents property of the item's storage unit. You can then use that information to make file-system-specific calls to open the file and process its contents into your part.
- To embed dropped non-OpenDoc data as a part, you treat it just as you would OpenDoc data. You follow the procedures described in Embedding a Single Part, specifying a cloning operation of kODClonePaste or kODCloneDropMove for the content storage unit of the dropped data. OpenDoc locates and binds a part editor to the data. That part editor itself then must obtain the item's HFSFlavor structure, access the file, read its data, and incorporate the data into the part's contents.

Completion of StartDrag

If your part is the source part of this drag, your call to the drag and drop object's StartDrag method completes after the destination's Drop method completes. You can now, based on the return value of StartDrag, confirm whether the operation was a move or a copy or whether it failed. Take these steps:

1. Whether it was a move or a copy, add an end action to the undo action history (see Adding Multistage Actions), so that the user can undo the entire transaction, from initiating the drag to dropping the data. If it was a move, note any applicable special considerations given in Handling Cut Data.

   If the completed drag was initiated as a drag-move but ended up being a drag-copy, and if it involved an embedded frame, follow the instructions listed in the table "Setting a Frame's In-Limbo Flag" in Undo and Embedded Frames (reset the frames's in-limbo flag to false).

2. If it was a move, delete the dragged content from your part and notify OpenDoc and your containing part that there has been a change to your content. See Making Content Changes Known.

3. Whether it was a move or a copy, call your source frame's SetDragging method once again, this time passing it a value of kODFalse to notify OpenDoc that the frame can once more be a drop target.

   Asynchronous drag and drop
   For future compatibility, OpenDoc provides the ODPart method DropCompleted. This method is called to notify the source part of the completion of a drag that it initiated. The method provides the same drop results that StartDrag returns for a synchronous drag.

Linking

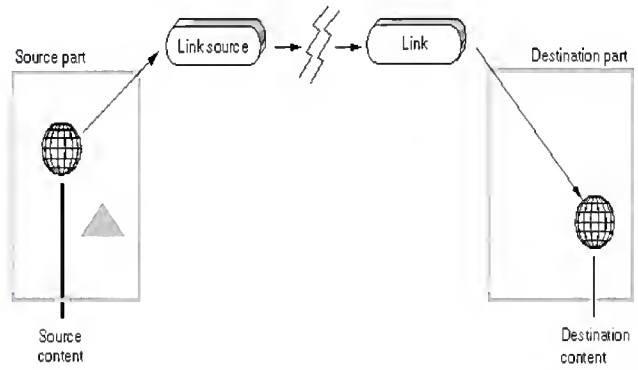
Linking is a mechanism for placing information into a part and allowing that information to be updated, based on changes to source information in another part or document.

Linking support in OpenDoc is a combination of event-handling code and storage code. It includes a set of notification calls that keep the transferred data synchronized with its source.

Some aspects of linking are closely related to the other data-transfer objects. Link specifications, used in writing data to the clipboard or drag and drop object, are described in Link Specification. The interactions between cutting and pasting and the preservation of link-related objects are listed in Transfer Rules for Links and Link Sources.

Link Concepts

Linking requires the cooperation of one or two parts with several linking-related objects. The following figure is a schematic illustration of the objects and data involved in linking. The figure shows a link between two separate parts (which can be in the same document or in different documents), although links can also occur within a single part.



The user creates a link by requesting it during a paste or drop operation. The part that contains the source of the information to be linked is called the *source part* . The content that is to be copied and sent to another part is called the *source content* or *source* . The source part

creates an object, the *link source* , that contains a copy of the source data; that copy is stored in the same document as the source part.

The part that contains the destination of the information to be linked is called the *destination part* . The content that is actually copied into the destination part is called the *destination content* or *destination* . The draft object of the destination part creates an object, the *link* , that is stored in the same document as the destination part.

The link-source object contains references to all link objects for which it is the source. The user is free to edit the source of a link; when a change occurs in the source part and the link needs to be updated, the source part copies the source content into the link-source object. The destination part then copies the content from the link object into the destination itself.

OpenDoc links are unidirectional; changes are propagated from the source to the destination only. You should not permit users to edit any link destinations that you maintain; such edits would be overwritten when the link is updated.

Whether it contains the source or the destination of a link, your part is responsible for defining the exact content that constitutes the linked data and for knowing what visual area it covers in your part's display. Linked data, whether source or destination, is part of your own content; you store and manipulate it according to your own content model and under your own control.

Link Update ID

Each link source and each link destination-that is, every region of content that is the source or destination of a link-must have an associated *update ID* . The source part determines the update ID associated with the link and updates the ID each time the source changes. The destination part stores the update ID of the link that was current at the time the destination was last updated through the link.

Whenever the content in your part changes-for instance, in response to a series of keyboard events-you need to assign a new update ID to all link sources directly affected by that change. Note that all link sources affected by a given change must have the s*ame* update ID. You obtain a new update ID by calling the session object's UniqueChangeID method.

When you update a link source, you can save the update ID in your source content and use it later to determine whether your source content and the content of the link-source object are identical.

A *circular link* or recursive link can occur when changes to a link's destination directly or indirectly affect its source. To avoid endless recursive updates, it is important that your part take these steps:

- Whenever your part updates a link destination that also is contained in a link source for which automatic updating is specified, it must propagate the update ID passed to it. Your part must update the affected source with the update ID passed to the destination, rather than calling UniqueChangeID.

  It is possible that your part containing the link destination is embedded in the link source content of another part. In that case, you must call your frame's content updated method so that the part owning the link source can be notified that your content has changed and be given the update ID of the change.

- Whenever a content change to your part results from an update to a link destination, your part must-when it calls its frame's ContentUpdated method-propagate the update ID passed to it.

- Whenever your part updates a link source manually (on explicit user instruction), it should not propagate an existing update ID. It should always call UniqueChangeID to get a new update ID.

Automatic and Manual Propagation of Updates

OpenDoc informs destination parts when the sources of their links have been updated. The update notification can be:

Automatic	Immediately (if the source is in the same document as the destination) or whenever the user saves the source document (if the source is in a different location
Manual	Only when instructed to do so by the user

The user selects whether updating is to be automatic or manual, either in the Paste As dialog box when the link is first created, or in the Link Destination Info dialog box after the link exists.

The user initiates a manual update by pressing the Update Now button in the Link Destination Info dialog box. Pressing the Update Now button in the Link Destination Info dialog box updates only the link destination from the link object.

To be notified automatically of changes to the source of a link, your destination part calls the link object's RegisterDependent method, supplying it the update ID of the last content it read from the link. Each link object maintains a private registry of dependent parts and calls their LinkUpdated methods whenever the link source changes. Your part should respond to this method call by updating its destination content

from the link object, as described in Updating a Link at the Destination.

If updating is to be automatic, your part should register for update notification when it first creates the link (passing an update ID of kODUnknownUpdate), and whenever your part reads itself from storage (passing the appropriate stored update ID from its link info structure). Be sure you are prepared to receive a notification when you call RegisterDependent, because your LinkUpdated method may be called before RegisterDependent returns. You can unregister and re-register as desired, but be careful not to register more than once at a time with the same link.

If your draft permissions are read-only, you should not register for update notification . Even if you do register, you do not receive notifications.

Updating a manual link without user instruction
If you maintain a link source set for manual updating, the state of your source content at any given time may be different from the contents of the link-source object. If another destination is added to the link, you must update the link-source object-even though the user has not explicitly requested the update-so that the new destination and all existing destinations will show the same content.

Automatic and Manual Updating of Link Source Content

OpenDoc allows owners of source parts to either specify that changes to link source content become visible as soon as the content changes (automatic), or only when explicitly requested (manually) by the source part. The source part calls the IsAutoUpdate method to determine the update mode. In the automatic mode (default), the part should update the Link Source Content Storage Unit (CSU) as soon as possible after the editing operation, which changed the content completely. In the manual mode, the parts should suppress updating the Link Source Content Storage Unit until the part's UpdateFrameLinkSource method is called.

Frame Link Status

All frames have a *link status* that describes the frame's participation in links. Parts are responsible for setting the link status of all frames that they embed; the link status indicates whether the frame is in the source of a link, in the destination of a link, or not involved in any link, or both. Parts can use this information to decide whether to allow editing of linked data, and whether to allow creation of a link within an existing link; see the second table in Transfer Rules for Links and Link Sources for a list of the possible combinations.

When to Change Link Status

In general, any time that you create a link involving an embedded frame or add an embedded frame to your part, you should set the frame's link status.

- When your part creates a link source, it should call the ChangeLinkStatus method of each of the embedded frames within the content area of the link source, passing it the value kODInLinkSource.
- If you add embedded frames to a link source, call ChangeLinkStatus for each new frame, passing it the value kODInLinkSource.
- If pasting and creating a link destination in your part involves adding embedded frames at the destination, call ChangeLinkStatus for each new frame, passing it the value kODInLinkDestination. (This situation may arise either during a paste-with-link involving embedded frames, or during an update of a link destination that includes embedded frames.) (This may occur either while performing a paste-with-link on embedded frames, or while updating a link destination that includes embedded frames.)
- Any time you embed a frame outside of any link source or destination that you maintain, set its status to kODNotInLink.
- If you break a link but keep the embedded data at the source or destination, set the status of each frame that was formerly linked to kODNotInLink (unless it is alos in a source; see next item).
- Destinations take precedence over sources. If one of your embedded frames is contained in a link destination that you maintain, and that destination is itself contained in a link source that you maintain, set the embedded frame's status to kODInLinkDestination. If you later break that link destination (but not its enclosing link source), set the frame's status to kODInLinkSource rather than kODNotInLink.

Note: If you do not support linking, you must nevertheless set the link status of your embedded frames (to kODNotInLink).

The ChangeLinkStatus method changes the value of the frame's link status, if necessary, and calls the LinkStatusChanged method of the

frame's part, so that the part can change the link status of any of its own embedded frames.

When you set the link status of a frame in any of these situations, you need to take into account only the links that your part maintains; you can ignore your own display frame's link status. OpenDoc automatically adjusts the link status of frames that you embed to account for your display frame's status.

You can examine the link status of a frame by calling its GetLinkStatus method.

The LinkStatusChanged Method of your Part Editor

Your own part's LinkStatusChanged method is called by the ChangeLinkStatus method of any of your display frames, whenever the display frame's link status is changed. This is its interface:

```
void LinkStatusChanged (in ODFrame frame);
```

Your implementation of LinkStatusChanged should iterate through all of your part's embedded frames, calling each one's ChangeLinkStatus method.

- The link status of embedded frames already in links that you maintain cannot change; sources will remain sources, and destinations will remain destinations. Therefore, you do not need to call ChangeLinkStatus for them.
- You can set the link status of embedded frames not involved in links that you maintain to be equal to your display frame's link status. However, you can also just set their link status to kODNotInLink, letting OpenDoc adjust their link status if necessary.

It is important to make this call to all of your embedded frames, so that they can in turn call their parts' LinkStatusChanged method to change the linke status of more deeply embedded frames, and so on.

You need not call the ChangeLinkStatus method of all of your embedded frames immediately upon receiving a call to ChangeLinkStatus. If you instantiate frame objects only when needed for display (see Lazy Instantiation), you can internally record which frames are affected, and the set their link status when you bring them into memory.

Note: If the user attempts to edit the content of any of your display frames whose link status is kODInLinkDestination, you should disallow the attempt. See Editing a Link Destination for details.

Content Changes in Embedded Frames

Because linked content can contain embedded frames, there must be a way for an embedded part to inform its containing part that its content has changed, so that the link can be updated.

The ContentUpdated Method

Any time a content change occurs in one of your part's display frames, you should follow the procedures described in Making Content Changes Known Part of the procedure is to call the frame' ContentUpdated method. The ContentUpdated method then calls the EmbeddedFrameUpdated method of your part's containing part, informing it that the content of one of its embedded frames has changed. If your display frame is involved in a link source maintained by your part's containing part, the containing part can then choose to update the link-source object with the new data.

(This ContentUpdated method is unrelated to the ContentUpdated method of ODLinkSource, discussed in Updating a Link at the Source. However, you pass the same update ID to both ContentUpdated methods when a content change to your part is entirely due to a content change to its link source.)

The EmbeddedFrameUpdated Method of your Part Editor

Your part's EmbeddedFrameUpdated method is called by the ContentUpdated method of any of your embedded frames, whenever the content of the frame's part has changed. This is its interface:

```
void EmbeddedFrameUpdated(in ODFrame frame,
                          in ODUpdateID change);
```

The method is passed a reference to the embedded frame and an update ID identifying the modification. You should respond by saving the Update ID and updating any link-source objects that you maintain that involve that frame (and whose updating is automatic). When updating the link source, pass it the Update ID that you received in the call to this method.

Your EmbeddedFrameUpdated method should also call the ContentUpdated method of your own display frames that contain the embedded frame. In this way, the change is propagated upward throughout the embedding hierarchy.

Link Borders

If your part contains the source or destination of a link, you are responsible for drawing an appropriate border around the linked content area when requested to do so.

Whenever you draw your part's content in a facet, first call the ShouldShowLinks method of the facet's window. If ShouldShowLinks returns kODTrue, draw borders around any link sources and destinations.

Link Info

The link info structure (type ODLinkInfo) contains fields that hold information about the nature of a link, such as the part kind of its intrinsic data, its creation date, and its update ID. Your part should allocate and maintain a link info structure for every link destination that it contains.

Here is the structure's definition:

```
struct ODLinkInfo
{
        ODType                          kind;
        ODTime                          creationTime;
        ODTime                          changeTime;
        ODUpdateID                      change;
        ODBoolean                       autoUpdate;
};
```

Linking and Undo

Just as the basic data-transfer actions (cutting, pasting, and dropping) should be undoable, so should their variations that involve linked data. The user should be able to undo (and redo) any of these actions:

- Pasting or dropping content and creating a link to its source.
- Pasting or dropping content that contains existing linked data.
- Deleting or cutting content that includes one or more link sources or destinations.
- Breaking a link at its source or its destination (through the Link Info dialog boxes).

When a link is created, the part receiving the data and creating the link destination adds a begin action and end action to the undo action history (see Adding Multistage Actions), whereas the source part adds a single action to the history when it creates the link source.

Edits to the source content of a link must also-like any edits to the content of your part-be undoable. However, updating a link source object from its source content does not need to be an undoable action.

- If the link source is updated manually, undoing or redoing changes to source content has no effect on the link source object. A manually updated link source always reflects the state of the source content at the last update.
- If the link source is updated automatically, changes to source content accomplished through an undo or redo action should cause you to update the link source as usual. (Be sure to use the update ID associated with the restored or redone content when updating.) Furthermore, automatic updating is not a user ation; since it is not performed by user command, it likewise cannot be undone.

Likewise, updating the destination content of a link from its link object does not need to be an undoable action:

- Editing the content at the destination of a link is not generally permitted. Because undoing or redoing an update to a destination would constitute editing the destination, you should never put the update action in the Undo or Redo stacks.
- You can, however, allow changes to the destination content that can be maintained across a link update, such as a style applied to the entire destination. Such non-editing changes can be undoable or ignorable, depending on your part's content model.

When you delete or cut content that includes a link source or destination, or when you break a link, follow the procedures outlined in Breaking and Cutting Links to make sure that you can undo the actions.

Manipulating Links

This section describes some of the basic procedures you follow in creating links, reading and writing linked data, and updating links.

Related information is described as part of general data-transfer considerations earlier in this chapter. Creating and removing link specifications are described in Link Specification and Removing a Link Specification from the Clipboard.

Both the source and destination parts of a link, or even separate destinations of a single source, may attempt to access linked data simultaneously. Therefore, many methods of the classes ODLinkSource and ODLink require that you provide a key before accessing the storage unit containing the content of a link. You obtain the key by first calling the Lock method of the link-source object or link object involved.

Creating a Link at the Destination

A link is created when the user decides-using the Paste As dialog box-to link rather than statically transfer data while performing a paste. The destination part-the part receiving the paste or drop-retrieves the link specification from the clipboard or drag and drop object and calls the part's destination draft's AcquireLink method. The source part's draft, in turn, calls the CreateLink method of the source part (the part that placed the data in the clipboard or drag and drop object).

If your part is the destination part that creates a link to pasted data from the data's source, you can use the following steps. It is assumed that you have previously called the ShowPasteAsDialog method of the clipboard or drag and drop object (see Handling the Paste As Dialog Box), and are ready to act on the results of the Paste As dialog box. For clipboard transfer, this means that you have already acquired the clipboard focus (see Acquiring and Relinquishing the Clipboard Focus).

1. Focus the clipboard or drag and drop storage unit on the link specification it contains. Use your draft's CreateLinkSpec method to instantiate the link specification, with the destination part as the part, and null for the data. Then call the link specification's ReadLinkSpec method to have it read itself from the storage unit.
2. This link-creation procedure should be undoable. If the data transfer for this link is a paste from the clipboard, add a beginning action to start a multistage transaction. That way, if the user decides to reverse the paste, both the paste and this link creation will be undone together. (Do nothing here if this data transfer is a drop.)
3. Pass the link specification to your draft's AcquireLink method to construct the link object from the link specification with 0 as the ID.
4. Create a link-info structure (type ODLinkInfo) to associate with the link destination, as described in Link Info. Initialize its update ID (to kODUnknownUpdate), set its creation time to the current time, set its modification time to the modification time of the link object, and give it a part kind and auto-update setting that reflect the user's choices in the Paste As dialog box.
5. Add this link object to whatever private list of link destinations you maintain. Store the information you need to associate this link object with its link-info structure and with the link-destination content in your part.

   If you have translated the data that now makes up your link destination, you also need to record the part kind that it was translated *from* . You will have to translate the data again for each subsequent update to the link, and you will need to know which part kind in the data to read and translate.

6. If the user has specified auto-updating in the Paste As dialog box and if your part has not already registered with this link object in connection with other link destinations, call the link's RegisterDependent method. Otherwise, manually update the link at this time by performing whatever actions your part's LinkUpdated method would perform (see Updating a Link at the Destination). To ensure that his link-creation procedure is undoable, take either of these steps:

    - If the data transfer for this link was a paste from the clipboard, add an end action to the action history at this time, to complete the two-stage action started in step 2.

    - If the data transfer for this link is a drop, you only need to add a single action to the undo action history. If the user undoes the drop, this link will be deleted also.

7. Call the ChangeLinkStatus method of any of your part's newly embedded frames that are within the linked content area, passing them the value kODInLinkDestination.

Never read the link content from the storage unit of the clipboard or drag and drop object when creating the link; always read link content from the link object's storage unit.

Every time the link is updated, your part (the destination part) must discard the data in the link destination and read it again from the link object. This process can entail discarding embedded parts and creating them anew from the updated link content. See Updating a Link at the Destination for details.

Your destination part must be able to draw a border around the link content area when asked to do so, as described in Link Borders. Parts embedded in linked data are not involved in the maintenance of the link; even in the case of a link destination that consists of a single embedded part, you draw the link border around the embedded part's frame.

Note that only the draft should call CreateLink; if your part is a destination part that needs a link to be created, you should call your draft's AcquireLink method.

Creating a Link at the Source

When the user decides to create a link to data that your part placed on the clipboard or drag and drop object, the destination part's draft calls the CreateLink method of your part (the source part). The draft passes back the data of the link specification that your part originally wrote when it placed the data on the clipboard or drag and drop object. This is the interface to CreateLink:

```
ODLinkSource CreateLink(in ODByteArray data);
```

When your source part's CreateLink method is called, the method should duplicate in a link-source object all of the content that your part originally wrote (or promised) to the clipboard or drag and drop object.

In your source part's CreateLink method, you can follow these general steps:

1. Examine your own structures to see if the link source object already exists. CreateLink may have been called as a consequence of another destination being added to this link source, as described in Replacing Link-Source Content in CreateLink.

    - If the link-source object does exist,it may not contain a complete set of promises or data. In that case, you need to write the remaining part kinds to it. You can do that by updating the link source in a particular manner. Skip to step 5.

      (If the link-source object does exist and does contain a complete set of promises or data, you could return the existing link-source object and take no further action. However, it may be simpler just to rewrite all part kinds than to test to see if you have a complete set.)

    - If the link-source object does not yet exist, continue with step 2.

2. Create a link source object by calling your draft's CreateLinkSource method.

3. Add this link-source object to whatever private list of link sources you maintain. Store the update ID and any other information you need to associate this link-source object with the link-source content in your part.

4. Call the ChangeLinkStatus method of any of your part's embedded frames that are within the linked content area, passing them the value kODInLinkSource.

5. Update the link source. Follow the steps listed in Updating a Link at the Source.

6. Creating a link source needs to be undoable. Add single action to the undo action history so that you can remove the link source if the user decides to undo its creation.

7. Unlock the link-source object.

8. Return a reference to the link-source object as the method result. (The caller is responsible for releasing the link-source object

when it is no longer needed.)

Replacing Link-Source Content in CreateLink

If your part's CreateLink method is called and the link-specification data provided describes an existing link source that you maintain, your part must ensure that all content kinds that you support are available at the destination. If your part writes promises to the link source, you can ensure that all kinds are promised by replacing the current content in your link-source object. The procedure for replacing is similar to the regular updating procedure, except that you pass an existing update ID to the link-source object's Clear method, and you do not call the link-source object's ContentUpdated method. See Updating a Link at the Source for details.

Updating a Link at the Destination

When a source part updates its link-source object and calls the object's ContentUpdated method, the link-source object notifies its associated link objects of the change (except for link objects in other documents, which do not receive the notification until the source document is saved). The link destinations receive notification of the change in this manner:

- If the user has selected automatic updating of a link destination associated with one of the notified link objects, the link object associated with the updated source immediately notifies the destination part of the changed by calling its LinkUpdated method.
- If the user has selected manual updating, the destination part is not notified. In this case, it updates the destination only on user instruction, as described in Automatic and Manual Propagation of Updates.

If your part contains the destination of a link, you can update the destination in your part's LinkUpdated method. This is the interface to LinkUpdated:

```
void LinkUpdated(in ODLink updatedLink,
                 in ODUpdateID change);
```

Your part will receive such a call in response to the user's request for an update in the Link Destination Info dialog box.

You can perform automatic updates in your part's LinkUpdated method, and manual updates in response to a user command in the Link Destination Info dialog box.

It is not always necessary to update immediately, during execution of LinkUpdated or when manually instructed to do so by the user. For example, if the destination has scrolled offscreen but is still registered as a dependent of the source, updating does not need to occur until (and if) the destination scrolls back into view. Your part editor can, if desired, generally perform link updates as a background task.

You can take the steps shown here to read the contents of a link object when updating your link destinations.

1. Call the link object's Lock method. If you cannot acquire the lock for the link it may be momentarily in use by another object; wait and try again.
2. Retreive your part's stored link-info structure for that link.
3. Access the link object's storage unit by calling its GetContentStorageUnit method. (If GetContentStorageUnit returns the error kODErrNoLinkContent, the last source update failed; do not update you destination at this time.)

   Incorporate or embed the updated destination data into your part from the link object's contents property:

   - If you are incorporating the linked data, follow the steps in Incorporating Intrinsic Content.
   - If you are embedding the linked data as a single part, follow the steps in Embedding a Single Part.
   - If you are translating the linked data before incorporating or embedding it, follow the steps in Translating Before Incorporating or Embedding Use information you recorded when you first created the link to know which part kind to read for translation.

   In any of these cases, the only differences from other data-transfer operations are that you call the Clone method of the link object's storage unit's draft, you specify a clone transaction of kODCloneFromLink, and you read from the link object's content storage unit.

   You must discard any previously embedded parts and frames, replacing them with new ones cloned from the link. Also, you must

connect the frames to your frame hierarchy, create facets for the currently visible ones, and change their link status by calling their ChangeLinkStatus method, passing the value kODInLinkDestination.

4. Update the link-info structure for the link, with a new update ID and new change time, obtained by calling the link's GetUpdateID and GetChangeTime methods.

5. Unlock the link.

6. Inspect your part's private structures to see if this change has affected the data of any link source in your part. (It can if this destination is within a source.) If it has, modify the source as shown in Updating a Link at the Source.

7. Notify OpenDoc and your containing part the there has been a change to your part's content; see Making Content Changes Known.

If your part maintains the destination of a link whose source has been transferred from one part to another, the new source part might not write its data using all of the same part kinds as the original source part. Your part, therefore, might not be able to obtain updates in the format that it expects. If that situation occurs, your part should read whatever part kind it can, or else break the link.

Note: This function is not supported for the current release of OpenDoc.

Updating a Link at the Source

The part maintaining the source of a link can rewrite values into the link-source object as necessary to update the link. If your part is the source part, take the steps listed here to update the contents of your link source (or to write it for the first time). The Automatic and Manual Propagation of Updates. describes how the user chooses whether updating should be automatic or manual. If the user has selected automatic (on-save) updating of the link source, perform this task whenver there is a change to the link-source content. If the user has selected manual updating of the link source, perform this task only on user command. The basic procedure for updating a link source is to first remove all content from the link source by calling its Clear method. By calling Clear, you ensure that all unneeded storage units are removed from the link, thereby saving space in your draft.

The Clear method deletes the contents property of the link-source storage unit and all its data. You need to add a contents property and values back into the storage unit after calling Clear, just as if creating the link source from scratch.

In summary, follow these steps:

1. Call the link-source object's Lock method, and then call its Clear method. If you cannot acquire the lock for the link source, it may be momentarily in use by another object; wait and try again.

    - If you are updating the link source because of content changes in your part, or if you are writing to the link source for the first time, or if this is a manual update, obtain a unique update ID and pass it to Clear.

    - If you are updating the link source because of an update to a link destination within your source content, pass the link destination's existing update ID (the one that was passed to your LinkUpdated method) to Clear.

    - If you are updating the link source because a change to an embedded frame caused your part to receive a call to its EmbeddedFrameUpdated method, pass the existing update ID (the one that was passed to EmbeddedFrameUpdated) to Clear.

    - If you are just adding new part kinds to an unchanged existing link source, pass the link source's existing update ID to Clear. (You must update the link if a new destination has been added and if the link source does not have a complete set of promises for all part kinds that you support.)

2. Access the link-source object's storage unit, create the contents property and the appropriate values, and write your updated source data. Write either data or a promise for all part kinds that you support, including standard part kinds. Update the frame shape annotation and any other needed annotation properties. If the data is intrinsic content-with or without one or more embedded parts-follow the procedure given in Writing Intrinsic Content.

    Specify a clone operation of kODCloneToLink. You are encouraged to write one or more promises instead of writing the actual data, to avoid placing unused content into the link source.

3. If you are just adding new part kinds to an unchanged existing link source, skip this step. Otherwise, after you have updated the data of the link-source object, notify it of the change, like this:

    - If the content change originated in your source part, or if this is the first time you are writing to the link, or if this is a manual update, call the link-source object's ContentUpdated method and pass it a new update ID (obtained from the session object's UniqueChangeID method), so that it can in turn send update notifications to all its link destinations.

    - If the content change occurred in a link destination contained within your link source, call the link-source object's ContentUpdated method and pass it the link destination's existing update ID (the one that was passed to your

LinkUpdated method), to avoid indefinite updating of circular links.

- If the content change occurred in an embedded frame and your part's EmbeddedFrameUpdated method was called, call the link-source object's ContentUpdated method and pass it the existing update ID (the one that was passed to EmbeddedFrameUpdated).

4. Unlock the link source.

In general, it is not necessary and not always desirable to perform automatic updates to link-source objects immediately in response to every content change. When content is changing rapidly, such as during text entry, it is reasonable to wait for a pause before updating any affected links.

Writing Linked Content to Storage

When your part writes itself to storage (see Writing a Part to Storage), or when it writes all or a portion of itself to a data-transfer object (see Writing to a Data-Transfer Object), it writes any linked data-whether source or destination-just as it would write any other content. In addition, however, it needs to write information for reconstructing the link objects and link source objects involved.

Writing Links in Externalize

There is no standard way to associate a link-source object or link object with a specific region of your content; how you create that association depends on the nature of your content. However, for all links your Externalize method and your methods that write to data-transfer objects need to write persistent references to the objects themselves, plus additional information. You should write the information in a format that is standard to your part kind; that way, any part editor that reads your part kind can reconstruct links from the stored data.

For the source of a link, you need to include, in addition to a persistent reference to the storage unit of the ODLinkSource object, the update ID associated with the linked content.

For the destination of a link, you need to include a persistent reference to the storage unit of the ODLink object, plus the information in the fields of the ODLinkInfo structure: the part kind, creation time, modification time, update ID, and auto-update setting of the linked data. Also, if your part has translated the linked data at the destination, you need to store the original part kind of the data (the kind before you translated it).

Your part may also store extra information for convenience. For example, because your part needs to be able to draw a border around linked data, it may store link-border shapes in a private structure.

Writing Links for Data Transfer

If your part writes a single embedded frame (without surrounding intrinsic content) to the clipboard or drag-and-drop object, and if that frame is a link source or link destination, clone the link or link source object into the data-transfer object's draft and add a property with the name kODPropProxyContents to the clipboard or drag and drop storage unit. In that property write a value that references the link or link source object, as well as the link info fields and other necessary information. Any destination part that understands the proxy format can then recover the link.

If you write linked content to a data-transfer object and the destination part does not support linking, the content may be transferred but the links will not be.

Reading Linked Content from Storage

When your part reads itself into memory, it may need to read in link-source objects and link objects as well. Likewise, when your part incorporates data that the user has pasted or dropped, your part may need to modify link sources or link destinations contained in that data.

Special care is necessary because OpenDoc can eliminate links during data transfer (see,for example, the first table under Transfer Rules for

Links and Link Sources). Therefore, you must perform the following two tasks when reading in data that includes links:

- Before you attempt to read in a link-related object, you must ensure that your persistent reference to it is valid.
- For every link-source object that you read in, you must ensure that your part is assigned as its source. The original source of the link may have been in another part, if the source content has been moved or copied.

Reading Links in InitPartFromStorage

Here are the steps you can take to handle linked data when reading your part (in InitPartFromStorage). The general procedure for reading a part is described in The InitPartFromStorage Method.

1. As you encounter a persistent reference to a link object or link-source object in the content you are reading, ensure that the reference is valid. Before trying to read in the referenced storage unit, pass the persistent reference to the IsValidStorageUnitRef method of the storage unit containing the content you are reading.
    - If IsValidStorageUnitRef returns true, Call your draft's AcquireLinkSource or AcquireLink method to read in the object.
    - If IsValidStorageUnitRef returns false, the link-source object or link object no longer exists, and your part should no longer consider the associated portion of its own content as linked. This is not an error situation, and you need not notify the user if it occurs.
2. For each link-source object you read, call its SetSourcePart method to associate it with your part. Add the link-source object to whatever private list of link sources you maintain, and associate it with the specific content in your part that is the source of the link.
3. For each link object you read, associate the link object with the specific content in your part that is the destination of the link. Add the link object to whatever private structures you maintain for link destinations.
4. For each link object you read, be prepared to register for update notification if updating is to be automatic. Either at this point or (preferably) once the linked data becomes visible or affects visible content, pass the last-saved update ID to the link's RegisterDependent method.

Reading Links for Data Transfer

Here are the steps you can take to handle linked data when reading (cloning) from the clipboard or drag and drop object. The overall procedure for reading from a data-transfer object is described in Incorporating Intrinsic Content.

1. After the clone operation completes successfully-assuming that you have followed the cloning steps listed in Cloning Sequence and have saved all object IDs returned by the Clone method-pass the ID of each cloned link object or link-source object to your draft's IsValidID method to determine if its object is valid.

    If IsValidID returns true, call your draft's AcquireLinkSource or AcquireLink method to read in the object. If IsValidID returns false, the link-source object or link object no longer exists and was not cloned.
2. For each link-source object you read, call its SetSourcePart method to associate it with your part. Add the link-source object to whatever private list of link sources you maintain, and associate it with the specific content in your part that is the source of the link. (That information is included in the contents property of the transferred data and is defined by the format of the part kind your editor is reading.)
3. For each link object you read, associate the link object with the specific content in your part that is the destination of the link. Add the link object to whatever private structures you maintain for link destinations. (That information is included in the contents property of the transferred data and is defined by the format of the part kind your editor is reading.)
4. If you are reading and embedding a single part that is also a link source or destination, there should be a property named kODPropProxyContent in the data-transfer storage unit. The property should contain a persistent reference to the link-source object or link object associated with the embedded part, plus any other needed information such as the fields of the link info structure. If you understand the format of the data in the property, read it and use it.

If you have incorporated an existing link source into your part, you may not be able to write all the data types (part kinds) to the link source object that its previous owner did. That is not an error; simply write the part kinds that you support.

Revealing the Source of a Link

Users need to be able to navigate from the destination of a link to the source content. When the user selects the Show Link Source button in the Link Destination Info dialog box the destination part calls the ShowSourceContent method of the link object. OpenDoc, in turn, calls the source part's RevealLink method. This is its interface:

```
void RevealLink(in ODLinkSource linkSource);
```

If your part contains the source of a link, your RevealLink method should display the part content that corresponds to the link source, following these general steps:

1. Using whatever part-specific method you choose, locate a display frame of the source content. If no suitable frame exists, open a part window to display the source.
2. Ensure that the frame displaying the source is visible in the window. Find its containing frame by calling its AcquireContainingFrame method, then call the RevealFrame method of the containing frame's part. (The procedure is unnecessary if your part opened up a separate part window for the link source, because the display frame, being the root frame for the window, is already visible.)
3. Activate the frame, using your part's normal activation procedure (see How Part Activation Happens). Scroll the content in the revealed frame, if necessary, to make the link source visible.
4. Select the content and draw a link border around it.

If your part is the part that needs to reveal an embedded frame containing a link source, you receive a call to your RevealFrame method. This is its interface:

```
ODBoolean RevealLink(in ODFrame embeddedFrame,
                     in ODShape revealShape);
```

Scroll your content, if necessary, to reveal the portion of the specified embedded frame containing the specified shape. If the embedded frame is not currently visible because your containing part has scrolled your display frame out of view, call your containing part's RevealFrame method, passing it your own display frame and an appropriate shape (such as the shape of the embedded frame to be revealed).

Editing a Link Destination

Because OpenDoc links are unidirectional from source to destination, the OpenDoc human interface guidelines restrict the editing that a user can perform within the destination of a link. Basically, changing of content in a link destination is not allowed.

It is the responsibility of the destination part to block attempts to edit its link destinations. However, the part that initially receives an editing event may be a frame embedded within, perhaps deeply embedded within, the destination part. If your part's display frame has the status kODInLinkDestination and the user attempts to edit content within it, your part should call the EditInLink method of your frame. OpenDoc in turn calls the EditInLinkAttempted method of the part that maintains the link destination that includes your frame. This is the interface to EditInLinkAttempted:

```
ODBoolean EditInLinkAttempted (in ODFrame frame);
```

The part that maintains the link destination should, in its EditInLinkAttempted method, take steps similar to these:

1. It should check that it does indeed maintain a link destination involving the specified frame. If not, the method should return false and exit.
2. It should present to the user an alert box that allows the user to forego the editing, break the link, or display its source. In any case, the part should not activate itself.
    - If the user chooses to forego editing, EditInLinkAttempted should simply return true.
    - If the user chooses to display the source, EditInLinkAttempted should follow the instructions listed in Revealing the Source of a Link, calling the ShowSourceContent method of the link object. If ShowSourceContent fails, display an appropriate alert box notifying the user of the failure to find the link source, and return true from EditInLinkAttempted.

- If the user chooses to break the link, EditInLinkAttempted should follow the instructions listed in Breaking and Cutting Links, and return true.

If EditInLinkAttempted returns true, then EditInLink returns true to your part (the part in whose display frame the edit attempt occurred). If the user has broken the link, your part's display frame now has a different link status and you can therefore allow editing of your content.

If EditInLinkAttempted returns false, then EditInLink returns false and OpenDoc cannot find the part that maintains the link destination or it cannot navigate to the link source. Your part should then put up its own simple alert box informing the user that editing in a link destination is not allowed.

If the user selects more than one link destination in your part and then attempts to edit the data, your part cannot even call EditInLink. It should then disallow editing and display a dialog box, suggesting that the user select a single destination.

Breaking and Cutting Links

Users can delete links in several ways. When you delete or cut content that includes a link source, or when you break a link at its source, your part relinquishes ownership of the link source object, as noted in the first table under Transfer Rules for Links and Link Sources. You should:

- Disassociate the link-source object from your part's content
- Set the link's source part to null by calling the link source's SetSourcePart method
- save a reference to the link-source object in an undo action, rather than releasing the link source
- Call the SetChangedFromPrev method of your draft to make sure that this change is saved when the document is closed
- Change the link status of any affected frames (see Frame Link Status) for a link that has been broken

It is not necessary at this point to update the link source to delete its contents. If the user subsequently undoes the action, you can reverse the steps listed in the previous paragraph. You must be sure to call the link source object's SetSourcePart method once again to reestablish your part as the link's source.

When you delete or cut content that includes a link destination, or when you break a link at its destination, you should:

- Disassociate the link object from your part's content
- Save a reference to the link object in an undo action, rather than releasing the link immediately
- Call the link object's UnregisterDependent method so that your part will no longer be notified when updates occur, if the link has been registered and if there are no remaining destinations of this link in your part.
- Call the SetChangedFromPrev method of your draft, to make sure that this change is saved when the document is closed
- Change the link status of any affected frames (see Frame Link Status), if the link that has been broken

Reversing these actions undoes the break or cut. Your part should hold the references to a cut or broken link source or link until your DisposeActionState method is called, at which time you can release the link or link-source object.

Transfer Rules for Links and Link Sources

This section summarizes the semantics of data transfer involving link objects and link-source objects. This discussion is for informational purposes only; if you follow the procedures presented earlier in this chapter, the correct semantics will always occur.

During some data-transfer operations, OpenDoc may delete or redirect link or link-source objects. Other operations are permitted by OpenDoc but should not be supported by your part. This section lists the basic data-transfer operations, in terms of:

- What happens when your part moves or copies linked data
- What happens when your part creates a link destination when pasting or dropping data that is already linked
- What configurations of links within links your part should support

OpenDoc enforces the behaviors listed in the following table when existing links or link sources and their associated intrinsic content are transferred by drag and drop or with the clipboard, according to the specified kind of cloning operation.

Operation	Object	Behavior
Copy (Drag-copy, or Copy followed by Paste)	Link source	Data is transferred; link source is not.
	Link destination	In same document: Data is copied; link destination is copied also. Across documents: Data is copiedl; link destination is not copied
	Source and destination of same link	Data is copied; link source and destination are both copied (but any uncopied destinations of the original source remain linked to the original source) the copied content are broken)
Move (Drag-move, or Cut followed by Paste)	Link source	In same document: Data is transferred; link source is transferred also Across documents: Data is transferred; link source is deleted (destinations within original document become unlinked)
	Link destination	In same document: Data is transferred; link destination is transferred also Across documents: Data is transferred; link destination is deleted
	Source and destination of same link	In same document: Data is transferred; link source and destination are both transferred Across documents: Data is transferred; any unmoved destinations of the link source become unlinked. Note: The move function is not supported in the current release of OpenDoc.

When pasting or accepting a drop into your part, you can incorporate or embed data that includes existing link sources or destinations and at the same time attempt to make the data a link destination. In that situation, OpenDoc transfers that data and creates a link destination, but does not maintain any link sources or destinations that were included in the transferred data.

You can create links within links; OpenDoc does not prevent you from creating or pasting the source or destination of one link within the source or destination of another link. However, only some configurations make sense, as shown in the following table.

You can use your display frame's link status (see Frame Link Status, as well as private link-status information about your own intrinsic content, to decide whether to allow the creation of a link source or destination in any given situation. In general, you cannot edit a link destination.

To create a new...	Within an	Recommendation

	existing...	
Link source	Link source	OK. Result is two separate link sources with partially or completely overlapping content.
Link source	Link destination	NO. When the destination updates, it is generally impossible to know which changes should become part of the updated source.
Link destination	Link source	OK. Updates to the destination simply become part of the source.
Link destination	Link destination	NO. Updates to either link destination will overwrite the other.

Semantic Events and Scripting

This is the seventh of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes the scripting support provided for your parts by OpenDoc.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. For additional concepts related to your part editor's run-time environment, see OpenDoc Run-Time Features.

Scripting support is one example of the use of the OpenDoc extension interface. Extending OpenDoc through its extension interface is described in general terms in Extending OpenDoc.

This chapter first summarizes the Open Scripting Architecture and OpenDoc scripting in general. It then describes the OpenDoc semantic interface, a set of classes and handlers you can take advantage of in implementing your support for scripting. The chapter then describes how to write and install:

- Semantic-event handlers
- Object accessors
- Object-callback functions, coercion handlers, and pre-dispatch handlers

This chapter concludes with a discussion of how to construct object specifiers and send semantic events from your part.

Note: The explanations in this chapter extend the discussions of OSA events, the OSA event object model, and the Open Scripting Architecture presented in the *Open Scripting Architecture Guide and Reference for OS/2* . It is assumed that you are already familiar with, or have access to, the descriptions of scripting found in that book.

Scripting and OpenDoc

Scripting is a powerful way to automate and customize programs. By adding scripts to a program and executing them, a user can simplify and automate tasks that might otherwise require extensive and repetitive human interaction. If the program supports it, a user might even be able to use scripts to change the meaning of existing commands. As a simple example, a user might customize the meaning of a "Save" command so that it updates a logging database every time a document is saved.

OpenDoc allows for scripting of s*emantic event*s , distinguished from user events in that they are high-level actions understandable by the user and by the program, but independent of individual user actions. Semantic events can be used to open, close, and save documents; embed parts; and manipulate part content and appearance.

OpenDoc supports your part editor's ability to receive and process semantic events through its semantic interface, a set of classes providing methods that you can use or override. The class ODSemanticInterface is a subclass of the extension class (ODExtension), described in Classes for Extending OpenDoc. You use the semantic interface class to set up and register semantic-event handlers and callback functions. (OpenDoc also supports your part editor's ability to create and send semantic events through another object, the messaging interface, an object of the class ODMessageInterface).

Any entity that understands the semantic interface of a part can send it semantic events. Script editors can convert scripting commands to semantic events and send them to any object in any part of a document. Parts can send semantic events to their containing parts, to embedded parts, to sibling parts, to linked parts-to any parts for which they can find an address. Other types of OpenDoc components can also send semantic events; spelling checkers, for example, can use them to operate on the content of text parts.

Scripting support in OpenDoc can be pervasive; if you take full advantage of it, literally every action a user can take may invoke a script. However, a range of options exists, and you can decide what level of scripting support makes the most sense for your parts.

Your part editor supports scripting through the OpenDoc semantic interface in much the same way as a conventional application supports scripting. Your part editor must provide an interface to its content objects and operations, and it must accept semantic events. The document shell passes semantic events to OpenDoc to deliver to your parts. Event targets are described by object specifiers, which refer to objects in your parts in terms of your published content model. The document shell needs information from your part editor to resolve these specifiers so that it can determine where to deliver the events.

Open Scripting Architecture

The scripting capability of OpenDoc is based on the Open Scripting Architecture (OSA). OSA is a powerful cross-platform messaging and scripting system that can support multiple scripting languages. OSA has three basic components:

- The events messaging system
- The events object model
- One or more scripting systems

This section gives a brief summary of these components. For complete documentation of OSA, see the *Open Scripting Architecture Guide and Reference for OS/2*

OSA Events

The system of semantic events at the base of the OpenDoc implementation of OSA is OSA events. OSA events are described in the events chapters of *Open Scripting Architecture Guide and Reference for OS/2* .

OSA events are messages that applications use to request services and information from each other. The application making the request constructs and sends an event to the application receiving the request. The receiving application uses an event handler to extract pertinent data from the event, perform the requested action, and possibly return a result.

OSA events constitute a standard vocabulary of actions that can be requested. OSA events are grouped into suites of related events. The OSA Event Registry: Standard Suites describes several commonly implemented suites. The most important suite to support for general scriptability is the Core suite, the suite that contains events common to many kinds of application software.

When sending an event, a requesting application can specify that the event apply, not to the receiving application as a whole, but to some element of the application's data. For example, an application can request the data of a particular row of a particular table in a particular report. The sender constructs an object specifier that it sends along with the OSA event to denote the exact element or elements that the event applies to. The *Event Registry* includes definitions of the kinds of objects recognized within each suite.

The Event Manager is the component of system software that manages the construction, sending, and processing of OSA events. It handles OSA events in general and performs event dispatching for conventional applications. With OpenDoc, events and the Event Manager function essentially as described in the *Open Scripting Architecture Guide and Reference for OS/2* ; however, there are some differences caused by the differences between parts and conventional applications:

- OpenDoc does its own dispatching of OSA events. This is to ensure that the event is directed to the correct part and that the reply is directed back to the part that sent the original event. That behavior is almost exactly the same as that provided by the Event Manager, except that the sender of the event no longer has access to the event's returnID parameter. Because this parameter is for OpenDoc's internal use only it is therefore missing from the Send method of ODMessageInterface

- OpenDoc itself implements some of the Event Manager functions. For those functions, you must use the corresponding methods defined by OpenDoc instead of directly calling the Event Manager. Cover methods for event functions are contained in the classes ODSemanticInterface (for calling, installing, and removing handlers), ODMessageInterface (for constructing and sending events), and ODNameResolver (for resolving object specifiers). Functions for which there is no corresponding OpenDoc method are available directly from the Event Manager. The following table lists the OSA Event Manager functions and their equivalent

OpenDoc methods:

OSA Event Manager Function (do not use)	OpenDoc Method (do use)
AEResolve	ODNameResolver::Resolve
AECallObjectAccessor	ODNameResolver::CallObjectAccessor
AECreateOSAEvent	ODMessageInterface::CreateEvent
AESend	ODMessageInterface::Send
AEInstallCoercionHandler	ODSemanticInterface::InstallCoercionHandler
AEInstallEventHandler	ODSemanticInterface::InstallEventHandler
AEInstallObjectAccessor	ODSemanticInterface::InstallObjectAccessor
AEInstallSpecialHandler	ODSemanticInterface::InstallSpecialHandler (plus individual InstallCallback methods)
AEGetCoercionHandler	ODSemanticInterface::GetCoercionHandler
AEGetEventHandler	ODSemanticInterface::GetEventHandler
AEGetObjectAccessor	ODSemanticInterface::GetObjectAccessor
AEGetSpecialHandler	ODSemanticInterface::GetSpecialHandler
AERemoveCoercionHandler	ODSemanticInterface::RemoveCoercionHandler
AERemoveEventHandler	ODSemanticInterface::RemoveEventHandler
AERemoveObjectAccessor	ODSemanticInterface::RemoveObjectAccessor
AERemoveSpecialHandler	ODSemanticInterface::RemoveSpecialHandler

At run time, OpenDoc handles semantic events by accepting the events and passing them to the appropriate part editor. There is no separate dispatcher object for semantic events; the OpenDoc dispatcher that handles user events also dispatches semantic events.

Event Object Model

OpenDoc relies on the event object model to specify individual elements of a part's content. The object model defines a hierarchical arrangement of *content objects* whose nature depends on the content model of the part. OSA events have object specifiers in their direct parameters to access individual content objects within a part. Part editors provide object accessor functions, used by the name resolver's Resolve method to resolve the object specifiers in OSA events that the parts receive.

The event object model used by OpenDoc is slightly different from that used by conventional applications. Here are the differences:

- The OpenDoc version of the object model has the ability to deal with multiple parts in a single document. It achieves this ability by defining a context for each event. The context for an OSA event in OpenDoc is the equivalent of the object model's default container; it is the outermost object in the object hierarchy defined by the direct parameter. The default container for an OSA event is typically represented by the application that receives the event. The context for an OpenDoc OSA event, by contrast, might be the document shell, or any of the parts in a document. (Before and after an event is handled, the context is always the document shell; during processing, however, the context may change multiple times.)

- In processing an OSA event, OpenDoc reverses the typical sequence from that used by the OSA Event Manager. The object specifier in the direct parameter of an OSA event is resolved before any event handler is called. The direct object may contain a description of the destination part, and that determines the context of the event. OpenDoc therefore replaces the object specifier in the direct parameter of the OSA event with the *token* returned from the resolution (see Returning Tokens) before calling a part's event handler.

- The Resolve method of the class ODNameResolver, which is a wrapper for the Event Manager AEResolve function, has a parameter used only by OpenDoc. The parameter, of type ODPart, specifies the part that is the context from which to start an object resolution. Also, unlike AEResolve, the Resolve method has no callbackFlags parameter, because callback flags in OpenDoc are specified on a part-by-part basis; they are set when the part calls the SetOSLSupportFlags method of the ODSemanticInterface class.

- Object resolution is possible for parts embedded within parts that do not support scripting. OpenDoc provides default object accessors and event handlers that allow a message to pass through a part that does not support scripting into an embedded parts that does. See OpenDoc Semantic Interface for more information.

- OpenDoc adds a noun to the OSA Event Registry. To represent a part (more strictly a display frame of a part), the registry defines the descriptor type 'part', for which you can also use the constant cPart.

- OpenDoc does not support calling the RemoveSpecialHandler function with the keyword keySelectProc in order to disable object-model support. Instead, making this call throws an exception.

Your part editor should implement object accessors as if your part were a stand-alone conventional application-that is, with your part itself as the context.

Scripting Systems

A scripting system completes the OSA implementation on a platform. It is through a scripting system that the user designs, attaches, and executes code to generate the semantic events that a scriptable part receives.

Any OSA-compliant scripting system can be used with OpenDoc.

OpenDoc allows the user to employ any available scripting system and even to switch among them during execution.

Part Content Model

To fully support scripting of your parts, you must construct a content model that includes a full set of content objects, accessible through semantic events, to allow a complete range of operations on your user-visible content. Your part editor must provide accessor functions to resolve external references to content objects, and it must also provide semantic-event handlers that implement content operations.

Developing a content model and implementing the functions for resolving object specifiers and handling semantic events gives great advantages. You increase flexibility and you provide user control over the basic functions of your part editor. For example, your parts will automatically be able to accept input from currently unavailable or unforeseen user interfaces, such as voice or pen or touch, that generate semantic events.

If you also factor your part editor-that is, if you separate your part editor's core date engine from its user interface-you increase its flexibility even further. Interface elements such as menus and dialogs, if reconstructed to invoke scripts or otherwise generate semantic events, will still be able to communicate directly with your parts. Factoring also facilitates making your parts recordable (see hdref refid=odrcrdl.).

Content Operations

The operations of a content model should be consistent with user actions. They typically include selection, creation, deletion, insertion, setting of properties, and so on. Low-level internal functions, such as piece-table functions for efficient text insertion or matrix inversion for fast graphics processing, are probably not appropriate as content-model operations.

Your content operations correspond to the semantic events you support. You implement a semantic-event handler for each operation.

Content Objects

The objects of a content model should be consistent with what the user sees and manipulates, regardless of your part editor's internal structures. Inside a text part, for example, a user may see lines, words, paragraphs, characters, and embedded parts. Those are typical content objects for such a part. Internally, your part editor might maintain run-length encoding arrays, line end arrays, and so on. However, because they are not presented to the user, they are not part of the content model. Likewise, a graphics part might have content objects like circles, rectangles, and lines, regardless of what internal mathematical descriptions it uses.

If a part supports embedding, embedded parts within it constitute a special class of content objects. The containing part can use content operations to manipulate the frames of embedded parts and perhaps some part-wide properties, but it cannot in general manipulate their contents-only the parts themselves can do that.

The OpenDoc default object accessors already provide this basic ability to access the properties of, and send semantic events to, embedded frames. If your part is a container part and needs capabilities beyond this basic level, your object model can include a named type of content object that represents an embedded part to which you can pass events and from which you can extract properties.

User selections

If you want to make user selections controllable through semantic events, you must describe them with object specifiers, defined in terms of your part's content model. In most cases, you should allow a selection to be specified both by description ("word 1 to word 5 of 'Gettysburg Address'") and by contents ("Fourscore and seven years ago").

Resolving Object References

Object specifiers in an event can refer to any content objects, including embedded parts. When it receives events, your part must provide accessor functions to allow those specifiers to be resolved. The accessor functions of a part return tokens, which are descriptors that identify the content objects or properties of those objects.

As an example of content-object resolution, consider a document whose root part is a text part. The part contains several embedded parts, including a bar chart named "sales chart". Suppose that the user employs a script editor (or possibly a scriptable tool palette) to send a command to change the bar chart to a pie chart. A Set Data event with the part name "sales chart" as the direct parameter goes to the OpenDoc message interface, which must resolve the object specifier in the direct parameter before dispatching the event itself.

The message interface asks the document shell to identify the part named "chart part". The document shell cannot, so the message interface asks the root part. The root part's object accessor recognizes the name of the chart part and passes back a token representing the part's frame. The message interface then asks the root part to identify the "chart type" property. The root part cannot, so it returns a token that causes resolution to be passed onto the embedded part itself. The message inerface asks the embedded part's object accessor to identify the "chart type" property. The accessor does and returns a token for it. The message interface inserts the token into the semantic event and dispatches the event to the embedded part's semantic-event handler. The handler then changes the chart to a pie chart.

Object accessor functions and object-specifier resolution are described in more detail in Writing Object Accessors.

Levels of Scripting Support

You can implement different levels of support for scripting in your part editor. Each successive level requires more effort but it gives the user greater flexibility and control over the functioning of your parts.

Scriptable Parts

If you make your parts *scriptable* , they will have at least a basic level of scripting support. To do that, you must create a content model for your parts with defined content objects (available through object accessors) and content operations (represented by semantic-event handlers) that are meaningful to the user. Then your part editor publishes a description of its content objects and operations and accepts semantic events:

- Your part editor publishes its list of content objects and operations by placing them in its terminology resource, a resource of type 'aete" that each scriptable part editor must provide.
- Your part editor accepts semantic events through its semantic interface, available publicly as an OpenDoc extension object. The semantic interface includes object accessor functions as well as semantic-event handlers.

Even among scriptability there are different levels of support. You can allow script access to only a few content objects and operations or to many of them. For your parts to be fully scriptable, semantic events must be able to invoke any action a user might be able to perform.

One advantage of making your part scriptable is that you can allow it to be used by other parts, even parts that you may not have developed yourself. For example, if your part is a text part and you allow script access to all of its text-style settings, the user (or your containing part or a sibling part) can easily format any of your text for any purpose.

Recordable Parts

If you make your parts recordable, the user's actions can be captured as a series of semantic events, converted to scripts, and replayed at a later time to reenact the actions. A recommended way to make your parts recordable is to completely separate the user interface from the core data engine of your part editor.

If your part editor handles every user-interface event by generating, sending to itself, and handling an equivalent semantic event, it can process all semantic events through a standard bottleneck and have the Event Manager record the actions. This use of a bottleneck not only allows recording but also enhances code portability.

The events that pass through the bottleneck should be just the set of actions that match your part's content model. These events need not follow exactly the inner design of your core data engine, but they should reflect the complete range of user actions.

Customizable Parts

If you make your part's interface customizable, the user can only invoke all actions through scripts, but can also change the nature of the actions themselves by attaching additional scripts that are invoked when the actions are executed.

To support customizable parts, you must make your parts scriptable, and you must also define content objects and operations for interface elements such as menus and buttons, or provide other ways to trigger scripts. You must provide persistent storage for any scripts that the user attaches to its interface elements.

Making a part fully customizable requires that the part editor allow attachment and invocation of scripts during virtually any user action. For the highest levels of customizability, all menu commands and all editing actions should be scriptable.

Before processing any user event or semantic event, the part editor must check whether a script attached to the part wants to handle the event. If so, you allow the script to run.

If your parts are recordable, they are already almost customizable. Because for recording purposes you typically check for the presence of and invoke scripts in a semantic-event dispatching bottleneck, in which case you achieve full customizability with little extra effort.

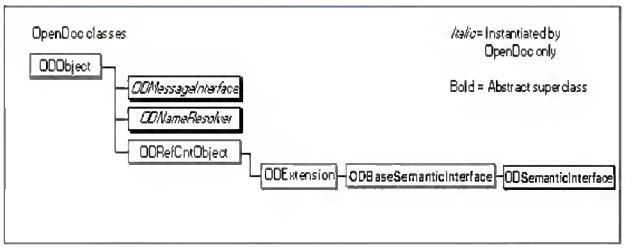
OpenDoc Semantic Interface

The class ODSemanticInterface defines the programming interface through which your semantic-event handlers and object accessors are called.

OpenDoc provides default semantic-event handlers and object accessors that allow it to access and send semantic events to scriptable parts embedded within parts that do not themselves support scripting. Even if your part does support scripting, you can use these defaults and build on their capabilities, rather than duplicating them.

Scripting-Related OpenDoc Classes

Several OpenDoc classes provide a structure for its support of semantic events and scripting. The following figure, a duplication of a portion of the OpenDoc class hierarchy (first figure in A Set of Classes, Not a Framework), shows the principal OpenDoc classes involved with scripting.

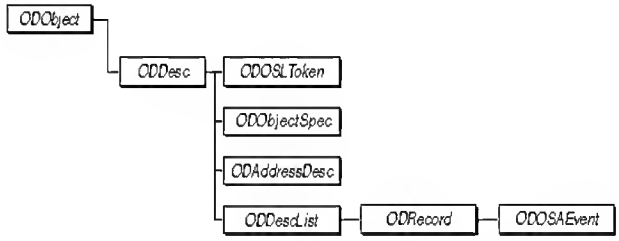


Three classes provide basic support for scripting in OpenDoc:

- The class ODSemanticInterface defines the programming interface through which script editors or other sources of semantic events communicate with your part. It is a subclass of ODExtension and is an abstract superclass.
- The class ODNameResolver represents the name resolver, a subclass of ODObject that is instantiated by the session object. It is used in resolving object specifiers; see Object Resolution.
- The class ODMessageInterface represents the message interface, a subclass of ODObject that is instantiated by the session object. It provides an interface through which your part sends semantic events to other parts; see Sending Semantic Events.

(The message interface is also the object through which OpenDoc sends semantic events to your part, although your part does not make any calls to the message interface in that situation).

OpenDoc also provides a set of scripting-related classes that are wrappers for OSA event descriptor structures. These classes, whose inheritance is diagrammed in the following figure, are analogous to the OpenDoc support classes shown in the last figure in A Set of Classes, Not a Framework.



The object-descriptor classes exist so that future versions of OpenDoc can support remote callbacks. They are mostly simple wrappers for structures, although some define few methods that you can call. Other than ODOSLToken, the structures that these objects wrap are all OSA events structures. Here is what each of them is for:

- The class ODDesc is a general wrapper for a descriptor structure (type AEDesc), the basic structure used for building OSA event attributes and parameters. ODDesc is a direct subclass of ODObject; the other descriptor classes are all direct or indirect subclasses of ODDesc.

  ODDesc provides methods for extracting and replacing the data of the OSA event descriptor it contains.

- The class ODOSLToken is a wrapper for an OpenDoc token, described in Returning Tokens. ODOSLToken provides a method for duplicating itself.

- The class ODAddressDesc is a wrapper for an address descriptor structure (type AEAddressDesc), a descriptor structure that contains a target address.

- The class ODObjectSpec is a wrapper for an object specifier structure (type typeObjectSpecifier), a descriptor structure that describes the location of one or more OSA event objects.

- The class ODDescList is a wrapper for a descriptor list (type AEDescList), a descriptor structure that is a list of other descriptor structures.

- The class ODRecord is a wrapper for an OSA structure (type AERecord), a descriptor list that can be used to construct OSA event parameters and other structures.

- The class ODOSAEvent is a wrapper for an OSA event structure (type AEEvent), a structure that describes a full-fledged OSA event.

The Default Semantic Interface

This section describes the semantic-event handlers, object accessors, and tokens that are provided as part of OpenDoc. They exist to make sure that script access is available to scriptable parts that are embedded within nonscriptable parts.

The default semantic interface provides basic access to the content and Info properties of embedded parts. When implementing scripting capabilities for your part editor, you can design your own handlers and accessors to build on, rather than duplicate, these capabilities.

Default Get Data and Set Data Event Handlers

To allow senders of semantic events to access certain basic information about any part in a document-regardless of whether that part or any of its containing parts is scriptable-OpenDoc provides two default event handlers. The default handlers respond to the Get Data and Set Data OSA events; the handlers are used for any part whose own handlers do not respond to those events.

The information manipulated by these handlers is the standard set of Info properties. These handlers cannot manipulate the content of any part. The event class of the handlers is kCoreEventClass, and their event IDs are kAEGetData and kAESetData.

You can rely on the default Get Data and Set Data handlers to provide script access to your parts standard Info properties. Such limited access does not really constitute scriptability; to provide script access to your part's intrinsic content or to other Info properties of your part or of parts embedded in your part, you need to write you own handlers.

Default Object Accessors

OpenDoc provides default object accessors to give scripts access to information within parts that do not support semantic events. Even if your part editor does support scripting, you can rely on these accessors to perform the following specific tasks. You need write object accessors only for other purposes, such as accessing your intrinsic content.

- **Part.** From the null container, a default accessor can return a token representing an embedded part. If your part is not scriptable, if it does not provide such an accessor, or if its accessor does not handle this task, the name resolver's Resolve method uses a default accessor to return a reference to a frame embedded within your part.

  The default accessor can resolve references to embedded parts specified by part name, part index, or part ID. Part ID in this case is not the part's storage-unit ID, but its persistent object ID, an identifying value used only for script access. A part's persistent object ID is unique within its draft and is valid across sessions. You can obtain a persistent ID for a part or a frame by calling the GetPersistentObjectID method of its draft; you can recreate a part or frame object by passing its persistent object ID to its draft's AcquirePersistentObject method.

  The token returned by this default accessor has the format described in Standard Embedded-Frame Token.

- **Property.** From the null container, a default accessor can return a token representing a standard Info property of the part that is the current context. If you do not provide such an accessor, or if your accessor does not handle this task, the name resolver uses this default accessor to return a reference to a standard Info property of your part.

  The token rturned by this default accessor has a private format. See the note "Default accessors require default handlers" at the end of this section for an explanation.

- **Wild card.** From a container of type cPart, a default accessor can return a swap token (see Returning a Swap Token), so that the name resolver can switch the context from the current part to a more deeply embedded one. If you do not provide such an accessor, or if your accessor does not handle this task, the name resolver uses this default accessor to change the context to the appropriate part embedded within your part.

If your part editor supports scripting but you do not want to duplicate the functions of these default accessors, you can write accessors only for tasks beyond the defaults: accessing objects in your part's intrinsic content, accessing Info properties that you have defined for your own part of for parts embedded in your part, altering the ordering/indexing scheme for embedded parts, and so on.

When one of your object accessors receives a token that it does not recognize as having been created by another of your own object accessors, it can simply return the OSA event error errAEEventNotHandled to the name resolver. The name resolver then attempts the resolution with the default accessors.

Default accessors require default handlers
Tokens created by your object accessors contain data in a format that your semantic-event handlers can understand, whereas tokens describing Info properties created by the default accessors are in a private format. Therefore, if you allow the default accessor to return tokens for the standard Info properties, you also need to allow the default Get Data and Set Data handlers to manipulate those standard Info properties. In that case, your own GetData and Set Data handlers need only manipulate your intrinsic content plus any custom properties you have defined for your part or for embedded parts.

Standard Embedded-Frame Token

To allow access to scriptable parts embedded within parts that are not scriptable, OpenDoc provides the default part accessor, as described in the previous section. That accessor returns a token in a standard format. The token has a descriptor type cPart, and its dataHandle field contains only a frame pointer and a part pointer (of type ODFrame and ODPart, respectively).

If your part is a container part and is scriptable, it must be able to support the creation of an embedded-frame token, either through its own object accessors or by letting the default part accessor construct the token. If you use a private format to describe an embedded-frame token, you must provide a coercion handler so that OpenDoc can coerce the token into the standard format when necessary.

Implementing your Semantic Interface

To be scriptable, your part must create a semantic interface that allows it to accept semantic events, handle them, and reply to them. It must:

- Provide semantic-event handlers for all OSA events appropriate to its content model.
- Provide object accessors for its content objects
- Provide other kinds of handlers (such as object callback functions and coercion handlers) as appropriate
- Make public, through a terminology resource, the events it handles and content-object types it recognizes

---

Writing Semantic-Event Handlers

You need to create a handler for every semantic event that your part editor recognizes. Your list of semantic-event handlers defines the content operations that your parts engage in and allow users to access.

How your semantic-event handlers manipulate your parts' content is entirely up to you. This section discusses only the process by which your handler receives and handles a semantic event. Installing your handler is described in Installation

A typical source of semantic events is a script engine or another part. The script engine or part generates semantic events and sends them to your document's document shell. This is how your part ends up receiving and processing a semantic event:

1. The identity of your part-the destination part-generally have been encoded in the object specifier in the direct parameter of the OSA event. Your part can be at any level of embedding in the document, and the encoding can be either explicit or implicit. For example, your part may be the implicit target when the event explicitly requests a custom Info property of one of your embedded parts.
2. The OpenDoc document shell receives the OSA event and passes it to the message interface object, by calling the message interface's ProcessSemanticEvent method.
3. The message interface object calls the name resolver's Resolve method for the event's direct parameter. Within the Resolve method, the object-resolution process may cycle several times (see Resolving Object References), until your part's semantic interface has been accessed and an object accessor of yours has returned a token that specifies your part or an object contained within it.
4. The message interface object calls the CallEventHandler method of the semantic interface to dispatch the event to your semantic-event handler. The CallEventHandler in turn examines the event class and ID to determine to which handler to dispatch.
5. Your part may have an attached script for any semantic event. If so, you execute the script at this point. If not, or if the script does not handle the event, or if the script handler passes the event on after processing, you dispatch the event to the semantic-event handler.
6. Your semantic event handler, on receiving the event, decides whether and how to resolve additional parameters. There are three general possibilites:
    - You read data from a location the parameter points to. This is nondestructive, and you can resolve the parameter and perform the task (possibly by sending the Get Data semantic event) even if it is from another part.
    - You overwrite data or move data to a location the parameter points to. This is a destructive action and you should perform it only on your own part. If you are acting on a list of items, for example, be sure that every item in that list represents an object in your own part before overwriting or moving data to any of them.
    - You ignore the parameter. If you choose to ignore a paremeter, do not retreive it from the event or attempt to resolve its object specifier.
7. Your event handler carries out its operation, returning a reply OSA event if appropriate.

    You fill out reply OSA events just as conventional applications do, as described in the *Open Scripting Architecture Guide and Reference for OS/2* . If the semantic event was sent by another part, the message interface object generates and keeps track of the return ID for the event, so that your reply can be routed back to the sender's reply event handler.

Writing Object Accessors

Object specifiers in an OSA event can refer to any content objects. If your part is to receive OSA events, it must provide accessor functions to

allow those specifiers to be resolved. Your accessor functions return tokens (see Returning Tokens) that your own semantic-event handlers can interpret. OpenDoc passes those tokens to your handlers when it resolves objects in the direct parameters of OSA events; your handlers themselves call the name resolver's Resolve method and receive the tokens for other OSA event parameters that are object specifiers.

If your scriptable part is also a container part, it must provide object accessor methods for objects that represent embedded parts, and it must be able to relinquish the context-that is, it must be able to hand off the object-resolution process to an embedded part (see Returning a Swap Token). OpenDoc includes default object accessors that provide the minimum capability for swapping context, but your part can replace or add to the capabilities of those accessors, if desired.

This section describes how your object accessor is called and which tokens it constructs and returns. The section also describes the default accessors provided with OpenDoc so that semantic events can be sent to parts embedded within containing parts that do not themselves support scripting.

Object Resolution

OpenDoc resolves object specifiers for content objects within parts much as the OSA Event Manager resolves object specifiers in conventional applications. However, there are some differences, including these:

- For semantic events sent from outside of a document, the document shell is the first handler of an object-specifier resolution. The shell does not handle events meant for any of the individual parts in the document, so in most cases it returns an error. To find the right part, OpenDoc then uses object accessor functions provided by the default semantic interface and by individual parts to obtain part-relative tokens (tokens for which the part is the context).
- OpenDoc reads parts into memory, if necessary, when interpreting object specifiers. For example, an embedded part referred to in a specifier may not be currently visible and thus not yet read into memory. To resolve the chain of objects further, OpenDoc may have to read in the part and then access an object within it.
- The callback flags used by the Resolve method (equivalent to the callbackFlags parameter of the OSA Event Manager AEResolve function) have a scope that is local to each part, rather than global to the object resolution process. Each part sets its callback flags by calling the SetOSLSupportFlags method of its semantic interface.

OpenDoc takes the following steps to resolve a reference to a content object. The process starts when the document shell (in the case of semantic events from outside the document) receives a semantic event and calls the message interface's ProcessSemanticEvent method, as described under steps 1 and 2 in Writing Semantic-Event Handlers.

A. Message Interface Calls Resolve

The message interface calls the Resolve method of the name resolver (class ODNameResolver), passing it the object specifier in the direct parameter of the OSA event (steps 1 and 2 of the figure shown at the end of these steps).

If the direct parameter does not exist, there may be a subject attribute in the event that takes the place of a direct parameter. A subject attribute is an object specifier that refers to the target part by its persisten object ID (see Sending Semantic Events). All events sent through Object REXX include a subject attribute, and all events sent through the Send method of the message interface include a subject attribute (unless the toFrame parameter of Send is null). The presence of a subject attribute allows a scripting system to record an event's targets even if it has no direct parameter.

If there is no direct parameter or subject attribute, or if the direct parameter is not an object specifier, Resolve is not called and the event goes to the document shell (and then to the root part if the shell does not handle it).

B. Resolve Locates the Proper Object

Not shown in the figure represented at the end of these steps is the fact that the Resolve method first accesses the document shell's semantic interface and gives it a chance to resolve the object specifier. If, as is typical, the event is targeted to a part rather than to the document shell itself, the default semantic interface passes resolution to the root part. The Resolve method then takes the following steps, first with the root part and then with the appropriate embedded parts until the specifier is finally resolved.

(Because OSA event objects exist in a hierarchy of containers, the Resolve method may make several cycles through the following steps, encountering several context swaps before identifying the specific object within a hierarchy).

1. Starting with the default container (this part's context), the Resolve method calls the AcquireExtension method of this part to get its semantic interface and thence the list of its object accessors. The Resolve method finds the accessor for the specified property or element and calls it (step 3 of the figure shown at the end of these steps).

2. The object accessor returns a token to the Resolve method, following the steps listed in Returning Tokens.

    - If the object is not an embedded part, the accessor puts into the token whatever information is needed to map the token to the right content object, and returns the token to the Resolve method.

    - If the content object represents an embedded frame as a whole, the accessor creates and returns a token that specifies the embedded frame.

      OpenDoc provides a default part accessor that performs this task if this part's object accessor does not, or if this part is not scriptable. See Default Object Accessors.

    - If the content object represents a directly accessible property of an embedded part-either a standard Info property such as the embedded part's modification date, or a custom property that this part may have defined, such as "is-selected"-the accessor creates and returns a token that specifies the requested property.

      OpenDoc provides a default property accessor that performs this task (for the standard Info properties only) if this part's object accessor does not, or if this part is not scriptable. See Default Object Accessors.

    - If the content object represents an object within an embedded part, or a property of the embedded part that this part cannot access, the accessor returns a special token (see Returning a Swap Token). At the next pass, the embedded part's list of accessors is used instead (step 4 of the figure shown at the end of these steps).

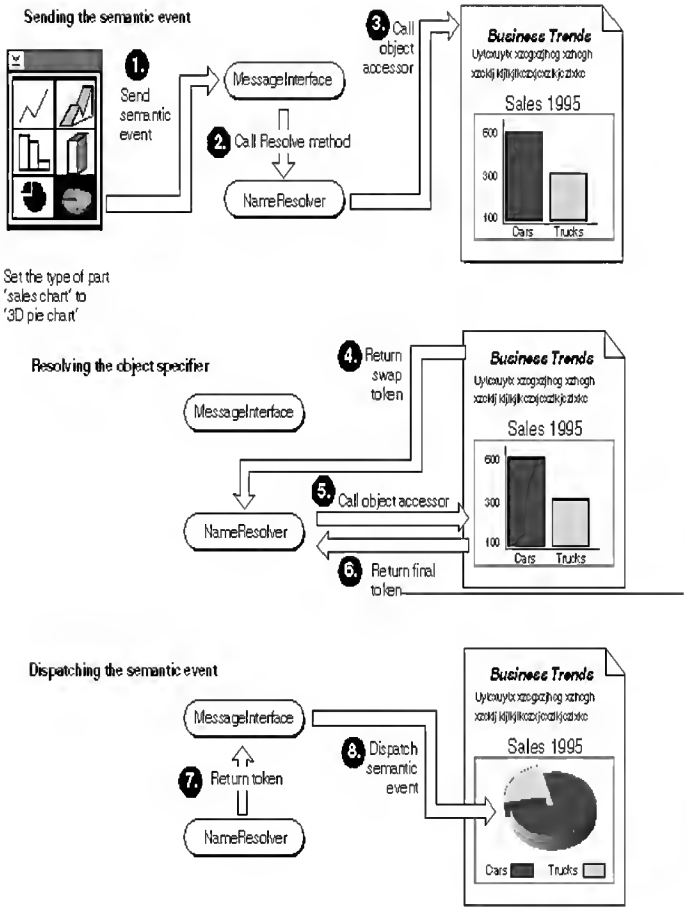
      OpenDoc provides a default wild-card accessor that performs this swap if this part's object accessor does not, or if this part is not scriptable. See Default Object Accessors.

3. The Resolve method finds the object accessor for the next property or element in the hierarchy of the object specifier and passes the returned token as the container to that accessor. That accessor, in turn, returns another token. This cycle continues, with context swaps occurring when appropriate, until the innermost element of the object specifier has been converted to a token and passed back to Resolve (steps 5 and 6 of the figure shown at the end of these steps).

C. OpenDoc Calls the Correct Event Handler

After resolving the object specifier, Resolve returns the final token to the message interface object (step 7 of the figure shown at the end of these steps). OpenDoc then passes that final token to the proper part's semantic-event handler, as the direct parameter of the OSA event (step 8 of the figure shown at the end of these steps).

(If the OSA event has a subject attribute but no direct parameter, OpenDoc calls the proper part's semantic-event handler but discards the token.)



Returning Tokens

In the OpenDoc version of OSA events, a token is a special descriptor structure, implemented as an OpenDoc object of type ODOSLToken, that a part uses to identify one or more content objects within itself. Your object accessor functions return tokens when they successfully resolve object specifiers. The structure of a token is not public, but it contains an OpenDoc object (of type ODDesc) that parts can access in order to extract or insert OSA event descriptor data.

OpenDoc hides the structure of the ODOSLToken and ODDesc objects; you cannot manipulate their fields directly. This privacy allows OpenDoc to store extra information that it needs inside a token, and it also ensures that OpenDoc's scripting support will be compatible with future distributed-object models.

When your object accessor needs to return a token, it modifies the ODOSLToken object that was passed to it by modifying the ODDesc descriptor object it contains. Your accessor can perform this task with a utility function or with methods of ODDesc itself.

1. The accessor calls the GetUserToken method of the name resolver to access the OpenDoc descriptor object (of type ODDesc) contained within the token that was passed to the accessor.

2. The accessor can optionally create a descriptor of type AEDesc and sets its descriptorType and dataHandle fields to store the information needed.

3. The accessor assigns the data to the descriptor object:

    - If it has created an OSA events (AEDesc) descriptor, the accessor can assign the AEDesc descriptor to the ODDesc object by using the function AEDescToODDesc (from the ODDesUtl utility library provided with OpenDoc). The accessor then disposes of the AEDesc.

    - If it has not explicitly created an OSA events descriptor, the accessor can assign the descriptor data to the descriptor object directly by calling its SetRawData and SetDescType methods.

The ODDesUtl utility library also provides the function ODDescToAEDesc, which allows you to extract (for inspection or modification) the AEDesc descriptor structure from an ODDesc object. The class ODDesc itself includes the methods GetRawData and GetDescType, which allow you to extract the descriptor data.

Your object accessors can verify the tokens passed to them by calling the name resolver's IsODToken method. The name resolver also provides the GetContextFromToken method, which allows your accessor to determine, for example, which display frame of its part contains the target of the event.

Returning a Swap Token

Sometimes your object accessor function is asked to access a content object (or a property that your part cannot directly access) from a object whose class is cPart-meaning that the requested item is something within a frame embedded in your part. In this case, the accessor must pass back a special token, called a swap token, to inform the name resolver of its inability to furnish the required token. Your accessor creates this token by calling the CreateSwapToken method of the name resolver to initialize the swap token, passing it a pointer to the embedded frame and a pointer to the part in the embedded frame. Your accessor then should simply return a value of noErr, taking no further action.

Upon receipt of the swap token, the name resolver changes the current context from your part to the embedded part, and retries the object access in that context.

If your part is a container part and is scriptable, it must support such context switches with swap tokens, either through its own object accessors or by letting a default accessor (see Default Object Accessors) perform the swap.

Note: OpenDoc reserves the descriptor type "swch", "part", and "tokn" for its own use. Do not use these values, or any value consisting solely of lowercase letters (such as "part"), in the descriptorType field of your own tokens.

Other Considerations

When you write an object accessor, note that, in interpreting object specifiers, "part X of doc Y" implies "part X of <current draft> of doc Y".

Your part can provide object accessors for document-wide user-interface elements, to be used when it is the root part of a document. For example, as root part it can provide accessors for window scroll bars or for document characteristics such as page size.

If your object accessor needs to know the frame through which your part was accessed-the frame that displays the part that is the current context-it can call the GetContextFromToken method of the name resolver. The value returned represents the most recent frame passed to CreateSwapToken.

Writing Other Kinds of Handlers

In addition to semantic-event handlers and object accessors, you can write other special-purpose functions and install them for use in interpreting semantic events. This section discusses OpenDoc issues related to object-callback functions, coercion handlers, and other kinds of handlers.

Object Callback Functions

Your part can provide object callback functions for the Resolve method to call where your part needs to provide extra information before object resolution can occur. You can use these functions, also called special handlers, for a variety of purposes:

- You can provide an object-counting function (CountProc) so that, when the object specifier involves a test, Resolve can determine how many elements it must examine in performing the test. You must provide this function if you want the OSA Event Manager to perform whose tests, that is, to resolve object specifier records of key form formTest without assistance from your part itself having to parse the whose clause and perform the counting test.
- You can provide an object-comparison function (CompareProc), which determines whether one element or descriptor record is equal to another. You must provide this function if you want the OSA Event Manager to perform whose tests without your part itself having to parse the whose clause and perform the comparison.

- You can provide a token-disposal function (DisposeTokenProc)-which overrides the Event Manager's AEDisdposeDesc function-in case you need to do any extra processing when your tokens are disposed of.
- You can provide an error-callback function (GetErrDescProc), provides a descriptor into which the Event Manager can write information about resolution failures.
- You can provide three kinds of marking-callback functions, which allow your part to use its own marking scheme to identify sets of objects:
    - A marking function (MarkProc) marks a set of objects.
    - An unmarking function (AdjustMarksProc) removes marks from previously marked object sets.
    - A marker-token function (GetMarkTokenProc) returns a token that can be used to mark a set of objects.

All callback functions in OpenDoc function exactly as they do with conventional applications, except for those minor changes:

- Each has an additional parameter, of type ODPart, to allow access to your part object from the callback function.
- Some parameters have different types from their conventional equivalents. For example, parameters of type AEDesc in conventional callbacks have types ODDesc or ODOSLToken in OpenDoc callbacks.
- OpenDoc callback functions return an error type of ODErr, instead of the OSErr type returned by callback functions for conventional applications.

You install object callbacks as described in Installing Handlers, Accessors, and Callbacks.

Object-callback functions (and whose tests) are described in more detail in *Open Scripting Architecture Guide and Reference for OS/2* .

Coercion Handlers

Coercion handlers are functions that convert data of one descriptor type into data of another descriptor type. Coercion handlers are common in OSA events. Some are provided by the Event Manager, others may be provided by your part editor. Any coercion handlers installed by your part editor are called only when your part is the context for the event. Normally, the document shell is the context, but there are two situations in which your part can become the context:

- When an object accessor installed by your part must be called during the resolution of an object specifier
- When your part's semantic-event handler is called

Coercion handlers are not chained by OpenDoc. That is, an embedded part does not inherit the coercion handlers of its containing part, and a root part does not inherit the coercion handlers of the shell.

Coercion handlers are described in more detail in the *Open Scripting Architecture Guide and Reference for OS/2*

Pre-Dispatch Handlers and Recording

A *pre-dispatch handler* is a function that is called whenever your document receives any event, before your part's handler for that OSA event is called. OpenDoc allows you to install pre-dispatch handlers and to specify whether or not you are currently using a given pre-dispatch handler.

For example, if your part is recordable, you can install a pre-dispatch handler to intercept the Start Recording and Stop Recording OSA events that are sent to the document shell. (OpenDoc does not automatically forward those events to each part in a document). Even so, your part may be read into memory and initialized after the user has turned recording on, in which case your pre-dispatch handler won't receive the Start Recording OSA event. Therefore, when your part initializes itself, it should also check with the OSA Event Manager to see if recording is on. If so, it can record its actions.

Pre-dispatch handlers are described in more detail, along with the keyPreDispatch constant, in *Open Scripting Architecture Guide and Reference for OS/2* .

Installation

Once you have an instance of the semantic interface, you need to install its components as described in this section.

Making the Semantic-Interface Extension Available

You must override the (inherited) part methods HasExtension and AcquireExtension so that they return the semantic interface object. The ODType constant that names the semantic interface extension is kODExtSemanticInterface; the constant is passed by callers of your HasExtension and AcquireExtension methods.

In implementing and interacting with the semantic interface, follow the rules for using OpenDoc extensions, as described in OpenDoc Extension Interface.

Installing Handlers, Accessors, and Callbacks

Your part editor's semantic interface is mainly a table of handlers that OpenDoc uses when processing a semantic event: You call methods of the semantic interface to install, remove, and call event handlers, object accessors and other special callback functions.

- You install semantic-event handlers by calling the InstallEventHandler method.
- You install object accessors by calling the InstallObjectAccessor method.
- You install coercion handlers by calling the InstallCoercionHandler method.
- You install object callback functions in either of two ways. You can call the InstallSpecialHandler method and pass it a parameter specifying the kind of function to install, or you can call function-specific methods such as InstallCountProc and InstallMarkProc. Constants for the parameter you pass to InstallSpecialHandler are defined by the OSA Event Manager, and have names such as keyAECompareProc.
- You install a pre-dispatch handler either by calling the InstallSpecialHandler method, passing it a parameter whose value is keyPreDispatch, or by calling the UsingPreDispatchProc method.

Other methods of the semantic interface allow access to and removal of these handlers, accessors, and callbacks.

You use the SetOSLSupportFlags method of your subclass of ODSemantic Interface to set flags that, during the resolution process, notify the name resolver's Resolve method of the kinds of object-callback support that your part editor can provide for resolution of object specifiers.

Installing System-Level Handlers

To install system-level semantic-event handlers, coercion handlers, object accessors and object callback functions, you can use the standard OSA Event Manager installation functions and set the isSysHandler parameter to TRUE.

In general installing system-level handlers is discouraged because of the difficulty of ensuring your part's availability to handle calls to them. If you do install them, be sure that you don't leave them installed after your part has closed. You should remove any installed system-level handlers in your part's ReleaseAll method or possibly in your part's Release method when the reference count goes to 0.

Making your Terminology Resource Available

When a scripting system first compiles a script that targets an OpenDoc document, it needs access to the terminology ("aete") resources associated with that document. The scripting system gains this access by calling the GetAETE method of the OSATerminology class.

Your part editor should install a terminology resource in the registration database by using the OSAInstallApplication function when it is first registered. When the GetAETE method is called with OpenDoc specified, all the terminology resources of all installed part editors, including the document shell terminology resource, are returned. Because the scripting system merges all terminology resources into a single composite resource for all parts and all documents, it is extremely important that your part editor avoid terminology conflicts with other part

editors. As a minimum, be sure to avoid conflicts with any terminologies defined in the current suites of OSA events.

Also, because the only available terminologies are those in the system-wide merged "aete" resource, your part editor cannot make its terminologies known dynamically; it cannot itself handle the Get AETE event.

Sending Semantic Events

Although the ability to send semantic events is not required for scriptability, OpenDoc provides specific support for it. This section discusses the OpenDoc-specific aspects of sending semantic events; for more information, see the *Open Scripting Architecture Guide and Reference for OS/2* .

The OpenDoc message interface object (ODMessageInterface) is responsible for constructing OSA events, getting and setting event attributes, parsing events, and sending events. This section describes how your part uses the message interface and the OSA Event Manager to send a semantic event.

To construct the OSA event, you call the CreateEvent method of the message interface object to create an OSAEvent object. You can then construct additional parameters and add them to the OSA event by calling the OSA Event Manager AEPutParamPtr or AEPutParamDesc function.

Constructing Object Specifiers

To send an OSA event to an OpenDoc part, the sender must use an object specifier that identifies the target part. The object specifier is typically in the direct parameter of the OSA event, although for events without direct parameters, OpenDoc places it in a subject attribute.

Conventional applications using the OSA Event Manager to send events to a part in an OpenDoc document construct the appropriate object specifier themselves. If your part sends a semantic event, you can use the methods described in this section to construct an object specifier. If you send a semantic event to your own part (as when recording), you follow the same procedures as when sending an event to another part.

Using CreatePartObjSpec

If your part sends a semantic event to another part, you can use the CreatePartObjSpec method of the message interface to construct an object specifier for the direct parameter of the event. You must also call the CreatePartAddrDesc method of the message interface to create an address descriptor that identifies the process (OpenDoc document) in which the destination part resides.

If you use CreatePartObjSpec, OpenDoc dispatches to the specified part directly, without calling the Resolve method of the name resolver. This dispatching is efficient and fast. However, you cannot inspect the contents of the object specifier constructed by CreatePartObjSpec or use it for recording, and you cannot use it as a component of another object specifier that you construct.

Using EmbeddedFrameSpec

You can also use the EmbeddedFrameSpec method of ODPart to help construct an object specifier. You can inspect the resulting specifier, you can use it for recording, and you can use it as the direct parameter or as another parameter in an OSA event. Object specifiers created through EmbeddedFrameSpec have forms such as "embedded frame 2 of embedded frame 1 of the root frame".

To construct an object specifier in this way, you call the EmbeddedFrameSpec method of the part containing the frame that is the target for the event. Here is the interface to EmbeddedFrameSpec:

```
void EmbeddedFrameSpec(in ODFrame embeddedFrame,
                       in ODObjectSpec spec);
```

A part receiving this method call should first call the EmbeddedFrameSpec method of its own containing part. It should then create an object specifier for the supplied embedded frame (using index number or any other appropriate identifying characteristic) and add it to the return value from its own call to EmbeddedFrameSpec. The final object specifier returned to the original caller thus describes the target frame in the

context of its document.

A root part receiving this call should return a null object specifier.

Constructing object specifiers with EmbeddedFrameSpec is most useful for situations in which a target part's position in the embedding hierarchy is more important than its specific identity. Note, however, that EmbeddedFrameSpec fails if any part in the embedding hierarchy from the target frame to the root frame is not scriptable or has not implemented the EmbeddedFrameSpec method. Also, an object specifier constructed by this method will become incorrect if the identifying characteristic of any frame in the hierarchy changes.

Using Persistent Object ID

The most reliable method for constructing an object specifier for a part is by using its persistent object ID. Use the persistent ID if you are constructing an object specifier and your call to EmbeddedFrameSpec fails, if you are sending an OSA event to a part with no display frames, or any time you need an object specifier that will be valid for a part regardless of its position in the embedding hierarchy.

Call the GetPersistentObjectID method of the target frame's or part's draft and use the return value in the object specifier. If your own frame is the target, as when recording, call your own draft's GetPersistentObjectID method and pass it your own display frame.

Sending the Event

To send the OSA event, you call the Send method of the message interface object. The message interface adds a subject attribute, representing the target part (your part if you are sending the event to yourself) to the event. The message interface also generates and keeps track of the return ID for the event so that the reply can be routed back to your part editor's reply event handler.

Extending OpenDoc

This is the last of eight chapters that discuss the OpenDoc programming interface in detail. This chapter describes how you can use or alter portions of the OpenDoc class library to enhance the capabilities of your part editors.

Before reading this chapter, you should be familiar with the concepts presented in Introduction and Development Overview. For additional concepts related to your part editor's run-time environment, see OpenDoc Run-Time Features.

This chapter discusses the following ways in which you can extend OpenDoc:

- By creating OpenDoc extension objects for your part editor, you can add programming interfaces to your parts for any purpose. The semantic interface support in OpenDoc, described in Semantic Events and Scripting is an example of the use of extension objects.
- By creating a settings extension, you can give users access to editor-specific settings through the Properties notebook.
- By creating specialized dispatch modules, you can define new kinds of user events that your parts can respond to.
- By creating specialized focus modules, you can define new categories of foci (shared resources) that your parts can acquire and exchange.
- By creating a subclass of ODTransform, you can extend the ways in which your part transforms the images it draws.
- By creating a shell plug-in, a modification of the functions of the document shell, you can add additional document-wide capabilities to OpenDoc.
- By patching (replacing) specific OpenDoc objects, you can modify some of the fundamental capabilities of OpenDoc.

OpenDoc Extension Interface

You can greatly extend the capabilities of your parts, in terms of fast processing of information or communication with other parts, if you use

the interface-extension capabilities of OpenDoc. The extension protocol allows parts or other OpenDoc objects to increase their capabilities by extending their programming interfaces. By using extension interfaces, your parts can communicate with other parts or other kinds of OpenDoc components in ways not possible with the standard OpenDoc programming interface.

The Semantic Interface extension to OpenDoc, described in Semantic Events and Scripting, is an example of the use of extensions to support scripting. Other kinds of extension interfaces can be especially valuable in those situations where scripting cannot provide enough integration or bandwidth.

You design, create, and attach such an extension to your part editor. At run-time, other parts can then access and use the extension interface, through calls to your parts.

Extension Objects

All subclasses of ODObject, including shapes, facets, documents, windows, frames, and parts, can be extended. An extension is itself an object, an instantiation of a subclass of ODExtension. Each extension object is related to its *base object* -the object whose interface it extends-by its extension name, an ISO type name. A caller accesses a base object's extension through that extension name. (*Base object* in this sense has nothing to do with inheritance; this book uses the term *superclass* to describe an ancestor in the class hierarchy). For example, the extension name for the semantic interface extension to a part is kODExtSemanticInterface.

Extensible objects create and delete their own extensions and manage the extensions' storage. If desired, a base object can share an extension object among multiple clients, perhaps using a reference-counting scheme to decide when to delete the extensions. A caller can query an extensible object to see if it supports a specified extension.

The ODExtension class itself has minimal functionality. It is designed to act as a superclass class for subclasses that implement actual extension interfaces. Every extension object knows what base object it is an extension of and forwards two kinds of calls to its base object: Release calls and calls to its own interface.

Using an Extension

To access the extension interface of an extensible part, a client, or caller, takes these steps:

1. It calls the part's override of its inherited HasExtension method to see if the extension is supported, passing the part (the base object) an extension name.

2. If the part has such an extension, the client then calls the part's (override of its inherited) AcquireExtension method to get a reference to the extension object. The part either creates the extension object or increases its reference count (if the object already exists) and passes the reference back to the client.

3. The client makes extension-interface calls directly to the extension object.

When the client has finished using the services of the extension object, it takes these steps:

1. The client calls the extension's (override of its inherited) Release method, to let the extension know that the client no longer needs it.

2. The extension, in turn, calls the (override of the inherited) ReleaseExtension method of its part (its base object) if its reference count has dropped to 0. The base object can then delete the extension.

   However, if the extension's base object has already been deleted and has called the extension's BaseRemoved method (see Closing your Part), the extension cannot call its base's ReleaseExtension method.

Implementing Extensions

Your part editors can implement any kinds of desired extension interfaces through this mechanism. The capabilities gained through extensions can be in almost any area. Examples include extensions to handle text search, spell-checking, linking, specialized text formatting, database access, and specialized graphics processing. In general, extension objects are best suited for tasks requiring high bandwidth or tight integration, for which scripting is not appropriate.

If you implement an extension object for your part, the extension should include a SOM constructor (somInit), a SOM destructor (somUninit),

and an initialization (InitExtension) method. The extension could also support a GetBase method, through which a client can obtain a reference to the extension's base object. (The ODExtension class provides a default implementation for GetBase).

Your part is the factory for its extensions, which are reference-counted objects. You must follow the procedures described in Factory Methods when creating, managing, and deleting extensions. Also, if your part's document is closed while it has extensions remaining in memory, your part's ReleaseAll method must call the extensions' BaseRemoved method.

An extension object must always be valid (attached to its base object) to be used. If a client tries to access an invalid extension, a kODErrInvalidExtension exception is generated. Any time after your part, as base object, calls its extension's BaseRemoved method, the extension is invalid. If you want to provide your own validation scheme for extensions, you need to override the ODExtension methods CheckValid, IsValid, and BaseRemoved.

Your extension interfaces can be private to your parts, or you can establish or follow public standards. CI Labs is the agency responsible for coordinating standard extension interfaces for parts. For information on existing extension interfaces, or to propose new interfaces, please contact CI Labs at the address shown in Cross-Platform Consistency and CI Labs.

The Settings Extension

The Properties notebook (see Selection properties) is accessed by the user from the Document or View menu and displayed by the Info object (ODInfo class). The information that OpenDoc displays in a part's Properties notebook consists of the part's Info properties. *Info properties* are those properties in a part's storage unit, separate from the part's contents property, that are intended to be visible to the user. They include properties such as creation date and modification date (which cannot be changed by the user) as well as name and part kind (which can be changed by the user).

The Properties notebook provides access to only the standard Info properties that all OpenDoc parts have. To define and allow access to Info properties specific to your part editor, you can create a *Settings extension* to display your own Properties notebook.

When you select the menu choice to display your part's properties, the ODInfo object queries your part for an ODSettingsExtension by calling the HasExtension method with the extension name kODSettingsExtension. If a part provides an extension, AcquireExtension is called and your parts notebook are displayed. If your part does not provide one, the standard notebook is displayed.

Your settings extension should be implemented as a subclass of the ODSettingsExtension class. ODSettingsExtension is itself a subclass of ODExtension.

Custom Event Types

You can extend OpenDoc's event-dispatching architecture to include new kinds of events by creating your own dispatch module.

Creating a Dispatch Module

The class ODDispatchModule is an abstract superclass. OpenDoc uses instances of a subclass of ODDispatchModule to dispatch certain types of events (such as keystroke events) to part editors. For normal program execution, you do not need to subclass ODDispatchModule or even access the existing dispatch module objects directly. Your interaction with OpenDoc regarding event dispatching is mainly through the dispatcher.

You can, however, provide for dispatching of new types of events or messages to your part editor by subclassing ODDispatchModule. For example, you could create a dispatch module that handled events from an exotic input device such as a 3D glove for a virtual reality game.

Patching the dispatcher
It is possible to use custom dispatch modules to patch the functioning of the dispatcher in relation to all event types, although that is in general not recommended. In most cases there is no need for such drastic alteration of OpenDoc functionality.

At run-time the dispatcher maintains a dictionary of installed dispatch modules, keyed by event type. When the dispatcher's Dispatch method is called, it looks up the dispatch module for the supplied event type and calls the Dispatch method of that module.

When the standard OpenDoc dispatch module transforms an event of one type (such as a mouse-down event in the menu bar) into an event of another type (such as an OpenDoc menu event), it passes the event back to the dispatcher for redispatching, by calling the dispatcher's Redispatch method. This redispatching allows your custom dispatch module to patch out or monitor the standard dispatch module for just those transformed events.

If you subclass ODDispatchModule, you need to implement a SOM constructor (somInit), a SOM destructor (somUninit), and an initialization method. Your initialization method should call (but not override) the InitDispatchModule method of ODDispatchModule.

Your dispatch module is responsible for performing the actual dispatching of events. The dispatcher calls your module's override of the Dispatch method, passing it the event information.

You install the dispatch module by calling the AddDispatchModule method of the dispatcher; you remove a dispatch module by calling the RemoveDispatchModule method of the dispatcher. The installation might occur during the initialization of your part (or shell plug-in, if you create one).

Your dispatch module should be installed just before your part will be handling events of the specified type, and removed right after your part is finished handling events of that type. If you are unable to add your dispatch module because another dispatch module already exists, you can still handle your events by doing the following:

- Use GetDispatchModule to obtain the address of the existing dispatch module
- Remove the existing dispatch module (using its address)
- Add your own dispatch module
- Handle your events
- Remove your dispatch module
- Add the previous dispatch module back

Using a Dispatch Module as a Monitor

You can also use a dispatch module as a *monitor* . In this case, the dispatch module is notified of events of its kinds but does not have to dispatch them. You might use a monitor in debugging, for example, to capture all events and display a log of them in a window.

You install a monitor with the dispatcher's AddMonitor method. For a given event, the dispatcher calls the Dispatch method for all installed monitors (of that event type) before calling the Dispatch method of the regular dispatch module for that event type. The dispatcher ignores the Boolean function result of the Dispatch method of all monitors; thus, unlike with normal use of a dispatch module, you can have more than one monitor for a single event type.

Custom Focus Types

You can extend OpenDoc's model for shared-resource arbitration by creating your own focus module, allowing your parts to recognize and negotiate ownership of new kinds of focus.

Creating a Focus Module

The class ODFocusModule is an abstract superclass. OpenDoc uses instances of a subclass of ODFocusModule to assign ownership of specific types of focus to part editors. For normal program execution, you do not need to subclass ODFocusModule or even access the existing focus module objects directly. Your interaction with OpenDoc regarding focus ownership is mainly through the arbitrator.

You can, however, subclass ODFocusModule for new types of focus ownership, perhaps related to new types of peripheral devices or new classes of shared resources. For example, if you provide an exotic input device such as a 3D glove for a virtual reality game, you could create a focus module that tracked the ownership of input from the glove.

You define a new kind of focus to be handled by that focus module by creating an ISO string that is the name of the focus. As an example, the ISO string that defines the scrolling focus is "Scrolling"; that and other currently defined foci are listed in the table shown in Focus Types.

Patching the arbitrator

It is possible to use custom focus modules to patch the functioning of the arbitrator in relation to all types of focus, although that is in general not recommended. In most cases there is no need for such drastic alteration of OpenDoc functionality.

If you subclass ODFocusModule, you need to implement a SOM constructor (somInit), a SOM destructor (somUninit), and an initialization method. Your initialization method should call (but not override) the InitFocusModule method of ODFocusModule.

Your focus module is responsible for maintaining the identities of the individual frames that own the foci that your focus module manages. You must implement the methods AcquireFocusOwner, SetFocusOwnership, UnsetFocusOwnership, and TransferFocusOwnership, all called by the arbitrator to request or change the owner of a focus.

You must also implement the methods BeginRelinquishFocus, CommitRelinquishFocus, and AbortRelinquishFocus, which your focus module uses in the two-stage process of relinquishing the ownership of a focus. The arbitrator calls these methods, and your focus module in turn calls the equivalent methods of the part that currently owns the focus.

You install a focus module by calling the RegisterFocus method of the arbitrator; you remove it by calling the UnregisterFocus method of the arbitrator.

The simple way
You can create a default focus module for any exclusive focus without having to subclass ODFocusModule. You simply pass the name of the focus and a value of kODNull (for the focus module object reference) to a RegisterFocus. The arbitrator then creates and returns a focus module for your focus that has properties equivalent to those of the existing OpenDoc focus modules.

Focus Modules for NonExclusive Foci

The arbitrator and focus modules allow foci to be *nonexclusive* , meaning that a given focus can have more than one owner. For instance, if video input is provided, a focus module for video focus might track the part or parts that are currently receiving video input.

If you create a focus module for nonexclusive foci, you must implement methods for testing exclusivity (IsFocusExclusive), and for allowing a caller to iterate over the owners of a focus (CreateOwnerIterator).

Custom Transform Objects

OpenDoc transform objects provide powerful transformational capabilities that are sufficient for most 2-dimensional drawing. With transforms, you can not only position your graphical objects, but you can also easily scale, rotate, and skew them. You can combine the operations of frame-internal transforms with those of facet-external transforms to achieve sophisticated effects with a minimum of code.

If you need to extend the power of transform objects even further, you can obtain the extra capability most efficiently by creating your own transform subclass. If you need to provide for nonlinear transformations (such as curved projections or sophisticated perspective effects), you can implement them as new methods and as overrides to the methods of ODBaseTransform, the superclass of ODTransform.

If you subclass ODBaseTransform, you must override at least the following methods:

```
Copy            InvertPoint         Reset
CopyFrom        InvertShape         TransformPoint
GetMatrix       PostCompose         TransformPoints
GetMATRIXLF     PreCompose          TransformShape
HasMatrix       ReadFrom            WriteTo
Invert
```

Your GetMatrix and GetMATRIXLF methods must throw the exception ODErrTransformErr. Your HasMatrix method must return kODFalse. Other methods should perform tasks appropriate to your transform.

For more information on matrices and transformations in two-dimensional drawing, you can consult any standard computer-graphics textbook, such as *Computer Graphics Principles and Practice* , 2nd ed., by Foley, vanDam, Feiner, and Hughes (Addison-Wesley, 1990).

Shell Plug-Ins

You can extend the capabilities of the document shell or add session-wide functionality to OpenDoc by implementing shell plug-ins. Shell

plug-ins are shared libraries, rather than subclasses of ODExtension. A shell plug-in is not associated with any particular part object.

Your shell plug-in must be installed on the user's machine when a document first opens, if it is to be used with that document.

You can create shell plug-ins for any of several purposes, including these:

- To install a dispatch module or focus module that will be available to all parts in the document
- To install an object (such as a spelling checker) into a name space, allowing it to be accessed as a service by all parts in the document
- To replace parts of OpenDoc with your own capabilities, from individual objects (see Patching OpenDoc) to entire subsystems (see Run-Time Object Relationships)

A shell plug-in has a single exported entry point to OpenDoc: its installation function. Your plug-in library needs to implement only that function, with this interface:

```
void ODShellPlugInInstall(Environment *ev,
                          ODDraft *draft
                          ODShellPlugInActionCodes *action);
```

The installation function must have exactly the specified name as well as the specified parameters and return value. You must ensure that the function name ODShellPluginInstall appears in your library's list of exported symbols.

The *draft* parameter specifies the document draft that is being opened. Use the *action* parameter to return a value specifying whether the document shell should maintain a connection to your shell plug-in library after the installation function completes.

The installation function must always return noErr, unless it cannot execute. For any ODShellPluginInstall function that returns an error, OpenDoc displays a dialog box to the user requesting that the shell plug-in be removed from the user's system.

Execution of a shell plug-in happens like this: Execution of a shell plug-in happens like this:

1. Whenever the user opens an OpenDoc document, OpenDoc launches the document shell. The document shell initializes itself and the session object, and gains access to the document's current draft.
2. The document shell then accesses each plug-in library and calls its ODShellPluginInstall function. The ODShellPluginInstall function performs the functions the plug-in is designed for: it installs custom focus modules or dispatch modules, it patches session-level objects, or it otherwise modifies shell functionality. ODShellPluginInstall then returns the appropriate OSErr value and exits.
3. After all plug-ins have executed, the root part of the OpenDoc document then opens the document window.

If it is to be executed, your shell plug-in file must be located in the OpenDoc Shell Plug-Ins folder on the user's system.

Patching OpenDoc

You can enhance or alter the functioning of OpenDoc in ways even more fundamental than implementing extension interfaces, dispatch modules, and focus modules. Because OpenDoc is modular and object-oriented, you can directly replace certain of its objects with your own versions.

OpenDoc allows you to patch any of its session-level objects. The session-level objects are those directly referenced by the session object, as shown in the figure presented in Session Object. They are represented by these OpenDoc classes:

- ODArbitrator
- ODInfo
- ODSemanticInterface subclass
  (The document shell's semantic interface)
- ODBinding
- ODClipboard
- ODDispatcher
- ODDragAndDrop
- ODLinkManager
- ODMessageInterface
- ODNameResolver
- ODNameSpaceManager
- ODStorageSystem
- ODTranslation
- ODUndo
- ODWindowState

This section discusses the mechanics of writing a patch to a session-level object, although it does not discuss the capabilities of the individual objects or the reasons for patching them. Before patching any object, be sure you are very familiar with its purpose and its interface. See information elsewhere in this book for more information.

Writing a Patch

Because your OpenDoc patch replaces a specific object with a known public interface, you should write it as a subclass of the class of object you are patching. You must override every method, and your overrides (other than somInit, somUninit, and your initialization method) should not call their inherited versions. If you want only partial replacement of OpenDoc's functionality, delegate to the patched-out object (the one you have replaced) those methods that you do not wish to change.

Immediately after creating your replacement object, your patch installer (see Installing a Patch) should call the object's initialization method (InitClassName). For example, if you are replacing the drag-and-drop object, you would call your replacement object's InitDragAndDrop method immediately after calling new to create it.

Your replacement object's initialization method should call the GetClassName method of the session object to get a reference to the current object and then store that reference in a field. For the drag-and-drop example, your InitDragAndDrop method would call the session object's GetDragAndDrop method, and store the returned reference in a field such as fOldDragAndDrop.

At the end of initialization, your InitClassName method should assign the replacement object as the current object. It should call the SetClassName method of the session object and passing itself (somSelf) as the new object reference. For example, if you were replacing the drag-and-drop object, you would make this call:

```
theSession->SetDragAndDrop(ev, somSelf);
```

Your destructor method (somUninit) should delete the patched-out object (the one referenced in the fOldDragAndDrop field, for example) before calling inherited somUninit.

If the object you are replacing has additional entry points besides those based on the session object's reference to it, you will need to patch those also. For example, the drag-and-drop object registers callbacks with the Drag Manager; your replacement object would have to re-register those callbacks to point to itself.

Installing a Patch

Once you have written your patch, you need to install it so that OpenDoc uses it in place of the original object. Depending on what your patching needs are, you can place your patch installer in either of two places.

- If the scope of your patch needs to be global (applied to all OpenDoc documents), make your patch installer a Shell plug-in (see Shell Plug-Ins). Install the patch within your plug-in's Install function.
- If the scope of your patch is confined to the document containing your part, you can use your part editor's Open method to install the patch. Only the root part of a document can install an OpenDoc patch.

When called, your installer instantiates and initializes the patch object.

Potential patch conflicts

Every time it opens a document, the document shell installs, in order, all shell plug-ins, and then the root part opens its window. This installation sequence can lead to patching conflicts. Because you do not control the order in which shell plug-ins are installed, you cannot ensure that your patch will not itself be patched out by a subsequently installed patch.

You can minimize the potential for conflicts by writing your patch correctly. Properly written patch objects always delegate everything but their specific functionality to the object that was previously in place. Consistently following this convention ensures the proper chaining of patch functionality.

Ultimately, the root part controls which patches remain, because it has the final opportunity to install its own patches.

OpenDoc Run-Time Features

This chapter concludes the programming portion of this book by discussing several aspects of the run-time environment that are of interest to you as a part-editor developer. The topics in this chapter are different from the programming-interface topics discussed in the previous eight chapters. Nevertheless, they are important to understanding how your parts function at run-time with OpenDoc.

- *Run-Time Environment* describes in general the run-time architecture of OpenDoc and how it affects part development.
- *Creating and Releasing Objects* describes how your part editor creates and releases objects (including how to be efficient about it), how it manages multiple references to objects, and when it should purge memory
- *Accessing Objects through Iterators* describes how your part can use iterators to access collections of objects
- *Binding* describes how your part editor is bound to a part at run-time, based on the kinds of data stored in the part and the kinds and categories of data supported by your editor.
- *Run-Time Object Relationships* summarizes and illustrates which OpenDoc objects in a document interact with each other for specific purposes.
- *Document Shell* describes how the OpenDoc document shell performs its functions and how that relates to your part editor's responsibilities

---

Run-Time Environment

The run-time model for the execution of part editors in the OpenDoc environment is different from that of conventional applications. This section discusses those differences in terms of the run-time process model, System Object Model run-time issues, and the part wrapper object. It also discusses name spaces, a run-time service provided by OpenDoc.

OpenDoc Run-Time Process

The OpenDoc document-centered approach requires a run-time process model significantly different from the classic application-centered process model. On personal-computer platforms, each process is usually owned by an application. This process provides the address space in which the application code executes, as well as the memory for every document opened by that application. The operating system assigns a document to a process based on the document's type, an indication of which application created it.

In OpenDoc, a document no longer has a single type, but is instead composed of many parts that may be of different types. Thus the document, not any application, is the owner of the process. The document manages standard tasks such as handling the event loop, managing files, printing, and interacting with system menus, dialog boxes, and so on. In OpenDoc, some of this behavior is provided by an executable called the document shell. The fact that individual parts in a document share this code is one reason why part editors can be significantly smaller than conventional applications.

The document shell has the following basic responsibilities:

- It creates and initializes the session object, which in turn creates and initializes the other session-wide OpenDoc objects, as shown in the figure presented in Session Object.
- It opens the document chosen by the user.
- It accepts events and passes them to the OpenDoc dispatcher.
- It handles certain document-wide menu commands.

The executable code of part editors is stored and accessed as DLLs independent of any process. (Note that part-editor code must be reentrant because several instances of a part in a single document may use the same editor). What the process provides is an address space for the contents of a particular document, along with any additional memory needed by the individual part editors. All state information used by any part editor is, with some minor exceptions, maintained in the processes of those documents that contain parts manipulated by the editor.

Dynamic Linking and System Object Model

Two cornerstones of the OpenDoc run-time environment are dynamic linking, which allows compound documents to be assembled at run-time, and a language-neutral object model, which facilitates run-time compatibility of parts and minimizes the recompilation required by changes to OpenDoc or to part editors.

Dynamic linking is provided by the base operating system. The object model used by OpenDoc is the System Object Model (SOM). This section introduces some run-time aspects of dynamic linking and SOM that are of interest to part-editor developers. (The particulars of writing SOM-based code are discussed in Developing with SOM and IDL).

Dynamic Linking

OpenDoc depends on dynamic linking to allow the appropriate executable code to be added to the run-time environment of a document. Before a document is opened, there is no way of knowing what part editors may be needed to manipulate its contents. Dynamic linking has the following capabilities:

- It allows an executable module to be loaded into memory once and then shared by multiple processes.
- It allows dynamically linked code to call code contained in another dynamically linked library.
- It gives dynamically linked code access to global variables.

SOM and Distributed Dispatching

System Object Model (SOM) objects are CORBA-compliant, meaning that they follow the language-neutral distributed standards of the Common Object Request Broker Architecture (CORBA), established by the Object Management Group (OMG), an industry consortium. This compliance means that objects compiled in different languages or with different compilers can communicate with each other.

More than that, SOM objects running on different machines can communicate with each other. This distributed dispatching may be a future capability of OpenDoc and is therefore supported in these basic architectural features:

- Almost all objects are instantiated by factory objects, rather than by constructors called by your part. See the table in Factory Methods for a list of the factory objects.
- The OpenDoc support for scripting is designed to allow for remote callback functions; see Scripting-Related OpenDoc Classes.

SOM Exception Handling

OpenDoc objects are SOM objects, and as such they follow the CORBA rules for handling exceptions. Every method call made to an OpenDoc object (including your part, as a subclass of ODPart) must therefore include an environment parameter (ev), a pointer to a value that can describe an error. For example, the CreateLinkSource method of ODDraft has the following prototype (in IDL):

```
ODLinkSource CreateLinkSource(in ODPart part);
```

The method takes a single parameter, of type ODPart. To use this method, however, a caller in C++ must supply two parameters:

```
MyLinkSource = MyDraft->CreateLinkSource(ev, somSelf);
```

If execution of the method results in an error condition, the receiver of the call (the draft object in this case) must place an exception code in the value pointed to by ev and return. The caller must therefore examine the ev parameter after every call to a SOM object, to see if an exception has been raised.

All OpenDoc methods that you call, as well as all public methods of your part editor that you write, must return errors this way. These are the implications for your exception handling:

- You must supply an environment variable with all method calls to OpenDoc objects.
- You must check the environment variable after the call returns.

The environment variable is passed along through a sequence of calls and can be used in calls to both SOM and C++ objects. Here is how you can use it correctly:

- If your SOM method calls another SOM method, it can simply pass on the environment parameter it has received. This is the usual case, since your part editor executes only in response to calls to its SOM interface, and thus should always have received a value for the environment variable.
- If your SOM method calls a C++ method that may in turn call a SOM method, your SOM method can pass the environment parameter on to the C++ method (if the C++ method was designed to accept it; see next bullet)
- If your C++ method is called by a SOM method and in turn makes calls to SOM methods, it is best to design it to accept an environment parameter that it can then pass on.
- If your C++ method (that does not itself receive an environment parameter) calls a SOM method, it can use a function provided by SOM (somGetGlobalEnvironment) to obtain a value for the environment variable.

For more information on the environment parameter and exceptions, see *SOMObjects Developer Toolkit Users Guide* and *SOMObjects Developer Toolkit Programmers Reference Manual* .

Any exception-handling scheme that you use must support this method of passing exceptions. See Exception Handling.

Part-Wrapper Object

Your part first initializes itself when OpenDoc calls its InitPart or InitPartFromStorage methods, as described in Initializing and Reading a Part from Storage. At that time, OpenDoc passes your part an object reference in the partWrapper parameter of InitPart or InitPartFromStorage. That reference is to a part-wrapper object, a private object that OpenDoc instantiates and uses to represent your part.

The part wrapper keeps other parts of OpenDoc from having direct pointers to your part object; its only function is to delegate methods of ODPart to your part. Using a part wrapper gives OpenDoc more flexibility in manipulating parts. For example, use of a part wrapper allows OpenDoc to switch editors for a part without having to close and reopen the part's document.

You must use that part-wrapper reference in all calls to the OpenDoc interface that require you to pass a reference to yourself. (Typically, if your part object keeps a field such as fSelf to hold a reference to itself, that field should contain the part-wrapper reference.) Specifically, you should never pass somSelf as a parameter to any OpenDoc method, except when accessing objects such as extensions and embedded-frame iterators for which your part itself is the factory.

Part-wrapper methods
The interface to ODPart includes three part-wrapper methods-IsRealPart, GetRealPart, and ReleaseRealPart-that are intended for the part-wrapper object only. OpenDoc calls these methods when associating your part with or disassociating it from its part wrapper. When you subclass ODPart to create your part editor, do not override these methods.

Name Spaces

Each OpenDoc session provides a run-time service allowing you to define name spaces. A *name space* is an object that maps data types to values. It consists of a set of string-value pairs, whose purpose is to associate the string (an ISO string) with the value (a four-byte unsigned value, which could be a pointer to code).

For example, OpenDoc uses name spaces to represent the preferences of the user for binding parts (whose part kinds are represented by ISO strings) to part editors (represented by pointers to editor IDs).

Name spaces are a general-purpose run-time registry mechanism. You can use them to define global spaces in which to share information with other parts. You can define as many name spaces as you wish; the OpenDoc *Name Space Manager* keeps a list (which is itself a name space) of all currently defined name spaces.

For example, you can use a name space to create palettes or other controls to be shared among several parts. Use an ISO string to identify the name space itself and ISO strings to identify each of the controls in encompasses. You then can get pointers to the controls (if the name space has already been instantiated) or instantiate the name space and provide pointers to the controls (if your part is the first in the session to use the name space).

If you publish the ISO strings that define your name space and its contents, the name space can be used publicly. If you do not publish the names, you can still use the name space for your own private globals.

Creating and Releasing Objects

This section discusses how your part editor allocates and releases OpenDoc objects, and how it should release unneeded memory when requested to do so by OpenDoc.

Factory Methods

Any OpenDoc object that you instantiate must be created by a *factory method* , a method of one class used to create an instance of another class. You should never use C++ new operator, for example, to create an object of an OpenDoc class (except for subclasses, such as ODExtension, through which you extend OpenDoc and for which your part is the factory).

The following table lists the factory methods you should use to instantiate any of the OpenDoc classes whose objects your part editor might ever need to create. The table list methods that create new objects as well as those that return previously created objects, whether currently in memory or stored persistently. Note that your own part editor has the factory methods for the classes ODEmbeddedFramesIterator and any of its own extension objects, including ODSemanticInterface. Note also that ODDraft has the factory method for ODPart and any of its subclasses, including your part editor.

Object to Be Created	Class and Factory Method
ODCanvas	ODFacet::CreateCanvas ODWindowState::CreateCanvas
ODContainer	ODStorageSystem::CreateContainer ODStorageSystem::AcquireContainer
ODDesc	ODNameResolver::GetUserToken
ODDocument	ODContainer::AcquireDocument
ODDraft	ODDocument::CreateDraft ODDocument::AcquireDraft
ODDragItemIterator	ODDragAndDrop::CreateIterator
ODEmbeddedFramesIterator (subclass)	ODPart::CreateEmbeddedFramesIterator
ODExtension (subclass)	ODObject(subclass)::AcquireExtension (usu.ODPart(subclass)::AcquireExtension)
ODFacet	ODFacet::CreateEmbeddedFacet ODWindowState::CreateFacet
ODFacetIterator	ODFacet::CreateFacetIterator
ODFocusOwnerIterator	ODArbitrator::CreateOwnerIterator
ODFocusSet	ODArbitrator::CreateFocusSet
ODFocusSetIterator	ODFocusSet::CreateIterator
ODFrame	ODDraft::CreateFrame ODDraft::AcquireFrame
ODFrameFacetIterator	ODFrame::CreateFacetIterator
ODLink	ODDraft::AcquireLink
ODLinkIterator	ODDraft::CreateLinkIterator
ODLinkSource	ODDraft::CreateLinkSource ODDraft::AcquireLinkSource
ODLinkSourceIterator	ODDraft::CreateLinkSourceIterator
ODLinkSpec	ODDraft::CreateLinkSpec
ODMenuBar	ODMenuBar::Copy ODWindowState::CopyBaseMenuBar
ODNameSpace	ODNameSpaceManager::CreateNameSpace ODNameSpaceManager::HasNameSpace
ODObjectIterator	ODObjectNameSpace::CreateIterator

ODOSLToken	ODOSLToken::DuplicateODOSLToken
ODPart (subclass)	ODDraft::CreatePart ODDraft::AcquirePart
ODPlatformTypeList	ODStorage::CreatePlatformTypeList
ODPlatformTypeListIterator	ODPlatformTypeList:: CreatePlatformTypeListIterator
ODSemanticInterface(subclass)	ODPart::AcquireExtension
ODSettingsExtension	ODPart::AcquireExtension
ODShape	ODShape::Copy ODShape::NewShape ODFrame::CreateShape ODFacet::CreateShape
ODShellPlugIn	ODPart::AcquireExtension
ODStorageUnit	ODDraft::CreateStorageUnit ODDraft::AcquireStorageUnit
ODStorageUnitCursor	ODStorageUnit::CreateCursor
ODStorageUnitRefIterator	ODStorageUnit::CreateRefIterator
	ODStorageUnitView::CreateStorageUnitRefIterator
ODStorageUnitView	ODStorageUnit::CreateView
ODTransform	ODTransform::Copy ODTransform::NewTransform ODFrame::CreateTransform ODFacet::CreateTransform
ODTypeList	ODStorageSystem::CreateTypeList
ODTypeListIterator	ODTypeList::CreateTypeListIterator
ODOValueIterator	ODOValueIterator::CreateIterator
ODWindow	ODWindowState::RegisterWindow ODWindowState::RegisterWindowForFrame
ODWindowIterator	ODWindowState::CreateWindowIterator

Reference-Counted Objects

The use of *reference-counted* objects is part of the OpenDoc memory management scheme. Reference-counted objects maintain a count of the current number of references to them; that is, they are shared objects that are aware of how many other objects are making use of them at any one time.

During an OpenDoc session, many objects are created. Because there can be a very complex relationship among objects, it might be difficult for a part or for OpenDoc to determine when it is safe to delete an object from memory.

Reference-counting is a way to determine when a run-time object can be deleted, so that valuable memory space can be reclaimed.

A reference count is 0 or a positive integer. A value greater than 0 means that at least one reference to the object currently exists, and thus the object must not be removed from memory. All descendants of the class ODRefCntObject and any of its subclasses (including frames, links, link sources, and parts) are reference-counted.

Each reference-counted object is created through a call to its factory method, the method (usually in another class) responsible for creating that object and initializing its reference count. For example, a draft object is created by calling the CreateDraft or AcquireDraft method of a document object. Likewise, a frame object is created by calling the CreateFrame or AcquireFrame method of a draft object. See the table in Factory Methods for a list of factory methods.

These are the reference-counted objects:

- ODContainer
- ODDocument
- ODDraft
- ODExtension
- ODFrame
- ODLink
- ODLinkSource
- ODMenuBar

- ODPart
- ODPersistentObject
- ODRefCntObject
- ODSemanticInterface
- ODSettingsExtension
- ODShape
- ODStorageUnit
- ODTransform
- ODWindow

When it is first created by its factory object, a reference-counted object has a reference count of 1. Thus, the frame object returned by a draft's CreateFrame method always has a reference count of 1, because that method always creates a new object. The frame object returned by a draft's AcquireFrame method, however, may have a reference count greater than 1, because that method may return a new reference to a preexisting frame.

Calling the Release method of a reference-counted object decrements its reference count; calling its Acquire method increments its reference count. Calling the Release method of a reference-counted object when its reference count is 1 causes its new reference count to be set to 0 and may result in the object being deleted from memory. It is an error to access an object whose reference count is 0.

Each reference-counted object stores its own reference count and returns it to callers of its GetRefCount method. When the object's reference count goes to 0, the object is responsible for notifying its factory object so that the factory object can delete it from memory. The factory object can choose to delete it immediately, or, for efficiency, keep it in memory until OpenDoc calls its Purge method when memory is needed.

Whenever your part editor writes a reference to a reference-counted object into a data structure, it should increment that object's reference count by calling its Acquire method. When your part editor is finished working with that object, it should call the object's Release method.

Because shape and transform objects are reference-counted and are widely used, it is important to remember always to release them instead of deleting them. Any method call with which you acquire a shape or transform must be balanced by a subsequent call to Release.

```
clipShape = facet->AcquireClipShape(ev, biasCanvas);
  . . .
clipShape->Release(ev);
clipShape = kODNULL;
```

Likewise, any method call that assigns a reference-counted object to another object increases its reference count. Therefore, you can immediately release your own reference to it.

```
newShape = facet->CreateShape(ev);
 . . .
facet->ChangeGeometry(ev, newShape, transform, biasCanvas);
newShape->Release(ev);
newShape = kODNULL;
```

Acquire versus Get
Methods whose names begin with *Acquir*e increment the reference counts of the objects they return; methods whose names begin with *Get* do not. Every call to ODDraft::AcquireFrame, for example, must be balanced with a call to ODFrame::Release. A call to Facet::GetCanvas, on the other hand, requires no corresponding call to decrement reference count.

As a reference-counted object, your part is responsible for implementing an override of the Release method and for calling the ReleasePart method of its draft object when your part's reference count goes to 0. Your part does not need to release all of its references or do any other shutting down or deallocation at that point; the ReleasePart method calls your part's destructor, which takes care of deallocation. However, your part could get rid of unneeded structures or services before calling ReleasePart. For example, a communications part editor may choose to close its driver as soon as its reference count reaches 0.

If your part has a reference count of 0, but calling your draft's ReleasePart method has not resulted in your part's destruction (perhaps because no purge has been performed), the draft object can retrieve and reuse your part. In that case, any structures or services you deallocated when the reference count went to 0 must be reallocated. Your Acquire method can perform those tasks for the case in which the reference count goes from 0 to 1.

Note that your part is destroyed after its ReleaseAll method is called, regardless of its current reference count, when its draft closes. See Closing your Part for more information.

Testing objects for equality
If you need to compare two existing OpenDoc objects for equality, don't simply compare their pointers. Instead, call the IsEqualTo method (defined in ODObject) of either object. IsEqualTo always gives the correct result, even in a distributed environment when comparing pointers may fail. However, be sure never to call the IsEqualTo method of a null object reference.

Handling Byte Arrays and Other Parameters

Many parameters to OpenDoc methods consist of references to objects or to other data that needs special attention in terms of allocating and releasing the storage associated with it. This section discusses parameter handling for byte arrays, strings, and objects.

To make the OpenDoc programming interface CORBA-compliant and capable of distributed execution, OpenDoc requires you to use a certain format when passing method parameters that point to buffers containing variable-length data.

OpenDoc defines the ODByteArray structure as a sequence of octets (unsigned 8-bit values). It consists of a buffer size field, a data length field, and a pointer to the buffer associated with the structure. All variable-length data is passed to or from OpenDoc with byte arrays. For example, the storage-unit methods GetValue and SetValue use byte arrays for the data being passed.

The caller of a method that takes variable-length data must place the data in a buffer pointed to by a byte array; the receiver of the data then retrieves the data from the buffer and uses it. Both sender and receiver must understand the underlying type of the data. For example, if your part calls SetValue, your part passes a structure of type ODByteArray, but the method understands the implied type of the contents of the buffer (a storage-unit value).

If you are the caller of a method that uses a byte array as a parameter, follow these rules for allocating and releasing the array:

- You as caller are responsible for allocating and releasing the storage of the byte array structure itself. You can allocate the byte array either on the stack or in the heap.
- If the byte array is an in parameter, you as caller allocate the storage for the data buffer the byte array points to, and you also release that storage after the method returns and you no longer need the data. (If the method needs to retain the data in the data buffer, it makes a copy).
- If the byte array is an out parameter or a function result, the method allocates the storage for the data buffer to which the byte array points; you as caller release that storage when you no longer need it.

All data passed by means of a byte array must be self-contained; for example, it can't contain pointers to data outside of itself.

If you are the caller of a method that uses a parameter that is a string (type ODISOString or one its equivalents, such as ODValueType) or an OpenDoc object, similar rules apply:

- If the string or object is an in parameter, you are responsible for both allocating and releasing its storage.
- If the string or object is an out parameter or a function result, the method allocates the storage and you must release the string or object when you no longer need it.

Purging

The OpenDoc document shell may monitor the use of resources within the process in which it executes. When resource usage goes above a predetermined value, it can ask other objects in its process to voluntarily surrender noncritical memory that they have been using.

You have a choice as to whether or not your part will handle the release of memory. If you decide to handle the release of memory, you need to override the Purge method to free the memory requested in the Purge call, as well as any caches or other noncritical buffers or objects. If necessary, your part can first write any data it desires to persistent storage.

Purging is the appropriate way to get rid of frames that you have created through lazy instantiation.

Lazy Instantiation

Your part does not necessarily have to maintain frame objects in memory at all times for all of its embedded frames and display frames. You may have many display frames or embedded frames, only a few of which are visible at any one time. In such a case, you might want to reduce memory use by creating frame objects only for those frames that are currently visible. Then, as the user scrolls through your part or resizes it, or brings other display frames of your part into view, you can create frame objects for frames that appear, and release frame objects from frames that disappear. This process, which could be called lazy instantiation, works like this:

- At initialization, your part can create frame objects for just the visible frames of its embedded parts, as described in Storing and Retrieving Embedded Frames.
- As each additional frame becomes visible through scrolling, resizing, or removal of obscuring content, you create the frame object by calling your draft's AcquireFrame method. OpenDoc then calls the DisplayFrameConnected method of the part (either your part or the embedded part) displayed in the frame.
- As each previously visible embedded frame becomes invisible, you can simply leave it in memory but marked as purgeable, or you

can call the frame's Release method.

- Maintain connections to your released frames by their storage-unit IDs. If a previously released frame becomes visible again, you can recreate it once again by again calling AcquireFrame. OpenDoc in turn calls the part's DisplayFrameConnected method once again, if necessary.

Accessing Objects through Iterators

The OpenDoc class library implements seven iterator classes, which you can use to access collections of OpenDoc objects such as facets, windows, and drag items. In addition, OpenDoc defines the interface for an iterator that you must subclass in all cases (ODEmbeddedFramesIterator), and one that you must subclass only if you create a focus module for a nonexclusive focus (ODFocusOwnerIterator).

All OpenDoc iterator classes share certain characteristics, and you can use them all in a similar fashion. The iterators that you implement should also function in the same manner. For example, all iterators have at least these three methods:

First	Begins the iteration: sets your position to the first element in the collection, and returns the element at that position.
Next	Advances your position in the collection by one, and returns the element at your new position.
IsNotComplete	Returns kODTrue if the element at your current position is valid; returns kODFalse if your current position is beyond the last element in the collection.

Some iterators also have these methods:

Last	Returns the final element in the collection.
Previous	Returns the element in the collection prior to your current position.

For most iterators, the First and Next methods return their collection items as function results; for some, the collection items are returned in output parameters.

Iteration is not complete until you have called First or Next and it has returned kODNULL. If you have just called Next and obtained the last member of the collection, subsequently calling IsNotComplete will return kODTrue because you can still call Next one more time (it will return kODNULL). After you have called First or Next and it has returned kODNULL, calling IsNotComplete will return kODFalse.

Some iterators allow you to traverse the collection in either direction (beginning-to-end or end-to-beginning). For those iterators, the direction is an invariant; once the iteration has begun, you can't change directions. If you are traversing the collection from the beginning to the end, you call First followed by a series of calls to Next; in this case you cannot call Previous. If you are traversing the collection from the end to the beginning, you call Last followed by a series of calls to Previous; in this case you cannot call Next.

You can set up an iteration in several ways. A common method is to set up a `for` loop that uses the IsNotComplete method to test for completion:

```
ODFrameFacetIterator* iter = frame->CreateFacetIterator(ev);
for (ODFacet* facet = iter->First(ev); iter->IsNotComplete(ev);
           facet = iter->Next(ev))
{
    . . .
    // Perform the task of the iteration
}
```

Or you can use a `while` statement to test for completion:

```
ODFrameFacetIterator* iter = frame->CreateFacetIterator(ev);
ODFacet* facet = iter->First()
while (iter->IsNotComplete())
{
   . . .
   // Perform the task of the iteration
   facet = iter->Next(ev);
}
```

A third possibility-if the Next method returns its item as a function result and if it consistently returns kODNULL for a nonexistent element-is to

use that method itself to test for completion:

```
ODFrameFacetIterator* iter = frame->CreateFacetIterator(ev);
while ((facet = iter->Next(ev)) != kODNULL)
{
   . . .
   // Perform the task of the iteration
}
```

OpenDoc iterators consistently follow these conventions:

- You can call Next without first calling First. If you do, your first call to Next works the same as calling First.
- You must call either First or Next before calling IsNotComplete. If you do not, IsNotComplete generates the exception kODErrIteratorNotInitialized.
- For an empty collection, First and Next return kODNULL (if that type of iteration returns its information in the function result) and IsNotComplete returns kODFalse.
- If IsNotComplete returns kODTrue, the last item obtained is valid.
- If IsNotComplete returns kODFalse (at the end of the iteration), Next will return kODNULL (if that type of iteration can return null values).
- If the collection is modified while an iteration is in progress, calling any of these iterator methods generates the exception kODErrIteratorOutOfSync.

Binding

Binding is the run-time process of assigning executable code to instance data. For parts, binding is the assignment of the correct part editor to a given part. For example, when the stored data of a part is to be brought into memory, OpenDoc binds a specific part editor to that data, loads the editor (if it has not already been loaded), and transfers control to the editor so that it can read in the data of the part. The resulting combination of part editor bound to part data constitutes the part object in memory, including both state and behavior.

On opening a document, OpenDoc binds editors to all parts that need to be displayed. During execution, OpenDoc binds part editors to part data when a part is read in or when its editor is changed. OpenDoc may unload part editors at various times, especially if memory is low. Also, when a part needs more memory, the document shell can unload the part editors used by inactive parts. Thus a given part editor may need to be bound and loaded more than once in a session.

The interface to the binding process is not entirely public. OpenDoc accomplishes it with the help of the binding object (class ODBinding). The *binding object* combines part-kind and part-category information provided by part editors, part-kind information stored with parts, and preferences specified by users to choose an editor for each part accordingly.

Information Used for Binding

This section describes the information your part editor must provide to make binding work correctly.

Part Kinds Supported by an Editor

For binding of parts to editors, these two kinds of information need to be available to OpenDoc:

- Your part editor must register information about the data formats it handles. OpenDoc uses this information to construct name space objects that it uses to bind part editors to parts.
- Your stored parts must include information about the data formats in them.

OpenDoc decides which editor to bind to a given part based on the part kind and part category of the data. Part kind is a typing scheme, analogous to file type, that can include specifications of data type, creator application, and even version. Part kinds are ISO strings such as

"text", "WaveDraw:Bitmap", or "AcmeDB:Kind:FlatDataBase". Part-editor developers define the part kinds that their editors use; each kind is a specific data format that the editor understands.

Tokenized strings
At run-time, OpenDoc and by part editors often manipulate ISO strings as tokens (short, regularized representations of the strings). Before passing ISO strings to OpenDoc, you therefore may need to convert them to tokens with the Tokenize method of the session object.

The fact that part editors can store multiple representations of their parts also means that they are not necessarily confined to reading and editing a single part kind. Your WaveWriter 3.01 Pro editor, for example, may be capable of reading and writing data of WaveWriter 1.0 ("WaveWriter:Kind:StyledText") and plain text ("text") format, as well as its own preferred format ("WaveWriter:Kind:IntlText").

At a minimum, your part editor needs to store only your own preferred part kind. A better alternative is to store your preferred kind plus one common standard public format. Beyond that, you should probably store additional formats only on user instruction, to keep your stored parts from being too large.

The method for defining the part kinds your editor supports differs among platforms. Your part editor registers the part kinds your editor can read and write. That information is converted at run-time into a name space object that the document shell can access.

Part Kinds Stored in a Part

Your part editor stores each of its parts in a format, or part kind, that your editor recognizes. But your part is not limited to one part kind per part; it can have multiple stored representations of its data in a single part. You store different representations in the kODPropContents property of your part's storage unit, as values with different part kinds, arranged in order of fidelity.

Fidelity refers to the faithfulness of a given representation to a part editor's native, or preferred, part kind. For example, assume that you store a part created with your WaveWriter 3.01 Pro part editor as three separate representations with the following part kinds: "WaveWriter:IntlText", "WaveWriter:StyledText", and "text". The highest-fidelity representation is "WaveWriter:IntlText", which represents the native format of your part editor; the "WaveWriter:StyledText" representation is an older, simpler format that lacks some of the latest WaveWriter features; and the "text" representation is plain, unformatted ASCII text.

Storing part representations in order of fidelity is crucial to ensuring that the best available editor is bound to a part.

Standard part kinds
To increase the chances of your parts being readable in all situations, you should if possible store at least one widely readable part kind, in addition to your editor's preferred part kind, every time you write your part to storage.

Part Categories Supported by an Editor

Part category is a typing scheme similar to part kind, except that it defines only a broad classification of the data manipulated by a part editor Like part kinds, part categories are ISO strings; they have designations such as "plain text", "3D graphics", "time", or "sound". (The full ISO string for the styled-text category, for example, is "OpenDoc:Category:Text:Styled").

Your part editor must specify the part categories corresponding to the part kinds it supports. Your editor registers, for each part kind that your editor can read and write, the part category or categories that the part kind belongs to. (A part kind can correspond to more than one part category; for instance, an unstyled text part could belong to both "plain text" and "styled text" part categories).

The following table lists the part categories that OpenDoc currently recognized and briefly explains the general kind of stored data each represents.

Part Category	Explanation
kODCategoryPlainText	Plain ASCII text
kODCategoryStyledText	Styled text
kODCategoryDrawing	Object-based graphics
kODCategory3DGraphic	3D object-based graphics
kODCategoryPainting	Pixel-based graphics
kODCategoryMovie	Movies or animations
kODCategorySampledSound	Simple sampled sounds
kODCategoryStructuredSound	Sampled sounds with additional

	information
kODCategoryChart	Chart data
kODCategoryFormula	Formula or equation data
kODCategorySpreadsheet	Spreadsheet data
kODCategoryTable	Tabular data
kODCategoryDatabase	Database information
kODCategoryQuery	Stored database queries
kODCategoryConnection	Network-connection information
kODCategoryScript	User scripts
kODCategoryOutline	Outlines created by an outliner program
kODCategoryPageLayout	Page layouts
kODCategoryPresentation	Slide shows or other presentations
kODCategoryCalendar	Calendar data
kODCategoryForm	Forms created by a forms generator
kODCategoryExecutable	Stored executable code
kODCategoryCompressed	Compressed data
kODCategoryControlPanel	Data stored by a control panel
kODCategoryControl	Data stored by a control, such as a button
kODCategoryPersonalInfo	Data stored by a personal information manager
kODCategorySpace	Stored server, disk, or subdirectory (folder) data
kODCategoryProject	Project-management data
kODCategorySignature	Digital signatures
kODCategoryKey	Passwords or keys
kODCategoryUtility	Data stored by a utility function
kODCategoryMailingLabel	Mailing labels
kODCategoryLocator	Locators or addresses, such as URLs
kODCategoryPrinter	Stored printer data
kODCategoryTime	Stored clock data

Part category is not included in the information stored with a part; only part editors store information about the categories of the part kinds they manipulate. OpenDoc uses that information at run-time to help the user define a default editor for each category.

User Strings

OpenDoc manipulates editor names, part kinds, and part categories as tokenized ISO strings. However, as noted in the following sections, there are several points at which the user can intervene in the binding process, and ISO strings are not appropriate for user display. OpenDoc requires that user-readable text be in the form of international strings that can be in any script or language.

Therefore, your part editor needs to provide, in its registration methods, user strings for its editor name, part kinds, and part categories so that

OpenDoc can display them to the user at the appropriate times. The user-readable name of your part editor (in English) might typically be close to, but not exactly the same as, its ISO string name and the kinds of parts it creates. For example, WaveWriter 3.0 might be the user-readable name of a part editor whose editor ID (ISO string name) is WaveCorp:WaveWriter 3.0 and whose native part kind is WaveCorp:WaveWriter:IntlText.

Binding Process

This section describes when binding occurs and how OpenDoc decides which editor to bind to a part, depending on which editors are available on the user's machine.

When Binding Occurs

Part binding occurs whenever your part is instantiated, typically when OpenDoc or another part calls the CreatePart or AcquirePart method of your draft. For example, binding can occur in the following situations:

- When a draft opens, OpenDoc binds a part editor to each visible part.
- As the user adds parts to an open document, OpenDoc binds part editors to the new parts.
- Drawing a part requires part binding if OpenDoc has not already bound an editor to the part.
- For a part to accept semantic events, OpenDoc must first bind its editor to it.
- When translation occurs, OpenDoc binds a part editor to the translated part.

Binding can thus occur in many different situations, not just when a part's document first opens. Your part editor needs to be able to function with any part to which it has just been bound, including a part that it has never edited before.

Binding to Preferred Editor

The most obvious binding for a part is to the editor that created it. A stored part may have, in its storage unit, a property of type kODPropPreferredEditor that specifies the editor that last edited the part. That editor is considered the preferred editor of the part. It may be the editor that originally created the part, or it may be an editor that was later bound to it. Every time a new editor is bound to a part, that editor becomes the part's preferred editor and remains so until a different editor is bound to the part.

When OpenDoc searches for an editor to bind to a part, it looks first for the preferred editor. If the preferred editor is present, OpenDoc binds it to the part.

If the preferred editor is not present on the user's system, or if there is no property of type kODPropPreferredEditor in the part, OpenDoc examines in turn each of the part kinds in the stored part, from highest fidelity to lowest, seeking to match that part kind with a part kind supported by an editor on the user's system. OpenDoc first examines the part's preferred kind (if the kODPropPreferredKind property exists), and then searches the remaining part kinds from highest fidelity to lowest (that is, from first to last in the contents property).

For each part kind under consideration, OpenDoc attempts to find an editor in this priority order:

- Default editor for kind
- Default editor for category
- Any available editor

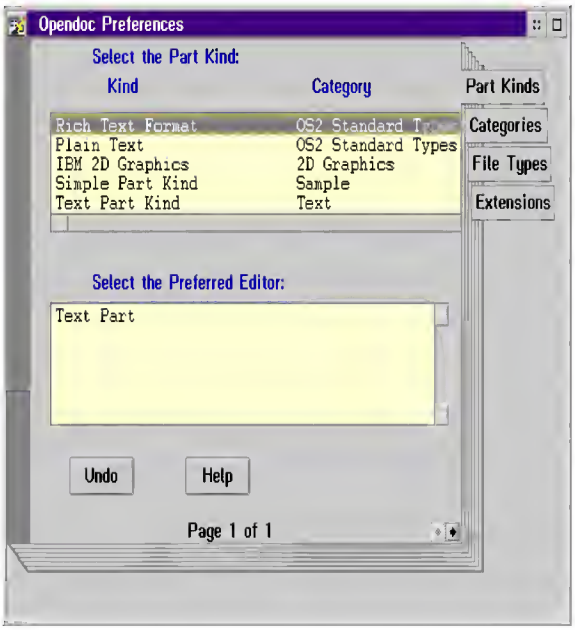
The following sections describe each of these steps.

Binding to Default Editor for Kind

After the preferred editor, the highest-priority binding for a part is to the *default editor for kind* , the user's chosen default editor for all parts of the part kind under consideration. For every part editor installed on a user's system, OpenDoc maintains a table that specifies the part kinds

for which that editor is the default editor.

By setting the default, the user chooses, for example, a single favorite text editor for processing any text part of a given part kind (when the part's preferred editor is not available). The following figure shows the Part Kinds page of the OpenDoc Preferences notebook, which displays the default editors chosen by the user.

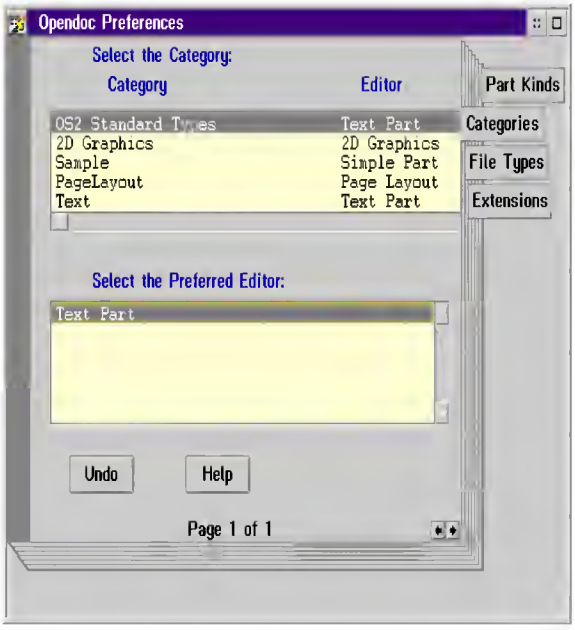


The "Select the Part Kind:" list box displays the kinds and their associated category, and the current default editor. The "Select the Preferred Editor:" list box lists only editors that can handle the selected part kind.

If, for the part kind under consideration, OpenDoc finds its default editor for kind, OpenDoc binds that editor to the part.

Binding to Default Editor for Category

If, for the part kind under consideration, there is no specified default editor for kind, OpenDoc looks for the default editor for the part category (or categories) of that part kind. As with the default editor for kind, OpenDoc maintains a list of the user's choice for default editor for each part category. By setting this default, the user chooses, for example, a single favorite graphics editor for editing any kind of bit maps (when its preferred editor and the default editor for its kind are not defined or not available). The Categories page of the OpenDoc Preferences notebook shows the categories and default editors (see the following figure).



The "OS/2 Standard Types" item shows that the user selected "Text Part" as the default editor for the "OS/2 Standard Types" category. As in the Parts Kinds page, only editors that can support a given category are displayed in the "Select the Preferred Editor:" list box. OpenDoc binds the "Text Part" to the part, if the editor can read the part kind, because it is the default editor for that category.

In addition to the Part Kinds and Categories pages, the OpenDoc Preferences notebook also allows a user to select a default editor for File Types and File Extensions.

Binding to Any Available Editor

If there is no default editor for the category of the part kind under consideration, OpenDoc then searches for any editor in the system that can read that part kind. If it finds such an editor, OpenDoc binds the editor to the part.

In the previous example, if the "Text Part" cannot read the part's kind, then OpenDoc looks for an editor that can.

If this attempt fails, OpenDoc repeats the entire process for each of the remaining (lower-fidelity) part kinds in the part. It looks first for the default editor for that kind, then the default editor for its category, and finally any editor that can read it.

Returning to the same example, suppose that the part has also stored a plain text version of its data. On the second round, OpenDoc locates the default editor for the plain text and binds it to the part. In this case, style information would be lost, but the user would still be able to read and edit the part.

This example illustrates the obvious advantage of having multiple stored representations of your part. The user has a better chance of being able to open and use some form of your part, even when your part editor is not available. Default editors exist only to allow the user to express a preference for an individual editor when a part's preferred editor is not present.

Binding to Editor of Last Resort

If there is no part editor on the user's machine that can read any of the part kinds stored in a part, the part remains unviewable and uneditable. However, OpenDoc still binds an editor to the part so that the part's document can be opened. This is the editor of last resort; it is always available.

On viewing the editor of last resort's properties, OpenDoc displays the part kind of the highest-fidelity version of the part. The user can use that information to determine what editor to purchase in order to read the part.

If the user attempts to open the part, OpenDoc displays a dialog box in which the user can specify a translation of the part to a part kind that can be read by an available editor, if such a translation is possible; see Binding with Translation.

The editor of last resort never modifies the part it displays; it does not change the part kinds or their storage order in the part's storage unit.

Binding with Translation

If the user's machine has no part editor that can handle any of the part kinds in a stored part, it can still be possible to find an editor-if any of the part's kinds can be translated into a part kind for which an editor does exist.

OpenDoc uses the translation object to give the user the opportunity to convert a part from a part kind that cannot be used to one that an existing part editor can read and edit.

Part editors initiate translation during data transfer (see Translation). OpenDoc initiates translation in the following situations.

- OpenDoc performs translation if the user specifies it after attempting to open a part displayed by the editor of last resort.
- OpenDoc performs translation if the user specifies it after attempting to open a document whose root part has no available editor.
- OpenDoc performs translation if the user specifies, in the Document Properties notebook. a part kind for the root part that requires translation. See The Document Shell and the Document Menu for more information.
- OpenDoc performs translation if the user specifies, in the Save A Copy dialog box, a part kind for the root part that requires translation. See The Document Shell and the Document Menu for more information.

To set up a translation to allow the opening of a part or document for which there is no editor, OpenDoc displays the Translation dialog box to the user. The dialog box contains pop-up menus from which the user selects a part kind to translate to and a specific editor to use with the translated part.

OpenDoc sets up the pop-up menu of part kinds in the Translation dialog box (and in the Document Properties notebook and Save A Copy dialog box) as described in Translation. Once the user picks a translated part kind, OpenDoc calls the translation object to perform the translation. Then OpenDoc binds the translated data to its new part editor and opens the part or document.

Consequences of Changing Part Editors

As a result of the binding process, your part editor can be bound to a part that it had not previously edited. (Binding to your editor is especially likely if the user has specified your editor as the preferred editor of that part's kind or category). This binding may occur in these situations:

- The part's document opens on a machine that does not have the part's previous editor, and your part editor can read its part kind or else the user translates the part to a kind that your editor can read.
- The user, using the **Paste as** command, pastes a part into a document and either specifies your part editor as the preferred editor or specifies that the part be translated to a part kind readable by your part editor.
- The user, in a part's Properties notebook or a document's Properties notebook specifies your part editor as the preferred editor or changes the part's kind (perhaps through translation) to a part kind readable by your part editor.
- The user makes a copy of a document with the Document menu's **Save a copy** command and specifies a part kind for the root part (perhaps requiring translation) that is readable by your part editor.

In each of these situations, once the binding occurs, the part's storage unit contains at least one part kind that your editor can read. However, the stored data in the part may not necessarily have exactly the same part kinds, in exactly the same order, that your editor would normally write into a part's storage unit. In this case, OpenDoc calls your part's ChangeKind method, passing it the new part kind.

OpenDoc can also call your part's ChangeKind method at times other than when your part editor is boud to a part. If your part editor supports more than one part kind and the user has specified (for example, in the Part Info dialog box or Document Info dialog box) that your current part's kind be changed to another kind that your part editor can read, OpenDoc calls your part's ChangeKind method and passes it the new part kind. This is the interface to ChangeKind:

```
void ChangeKind (in ODType kind);
```

Your part editor needs to start manipulating the part's data in the new format. Your ChangeKind method must store the data in an order that reflects your own fidelities. It should also make sure that the new part kind is specified in the part's kODPropPreferredKind property.

For example, suppose that your WaveWriter 1.1 part editor supports only "WaveWriter:StyledText" and "text" formats. Suppose further that your editor is bound to a part whose highest-fidelity part kind is "WaveWriter:IntlText" (perhaps originally created with the WaveWriter 3.0 editor) but that also contains a "WaveWriter:StyledText" representation. In this case, you need to make sure that "WaveWriter:StyledText" is specified as the preferred kind when you subsequently write the part. You could also include a "text" representation, which must be stored after the preferred kind. (If your part editor supports it, you can optionally write a higher-fidelity version than the preferred kind; if you do, you must place it before the preferred kind in your storage unit).

Note: These modifications do not occur if the editor of last resort is bound to a part. It never modifies the part it "displays"; it does not change the part kinds of the values, or their fidelity ordering, in the part's storage unit.

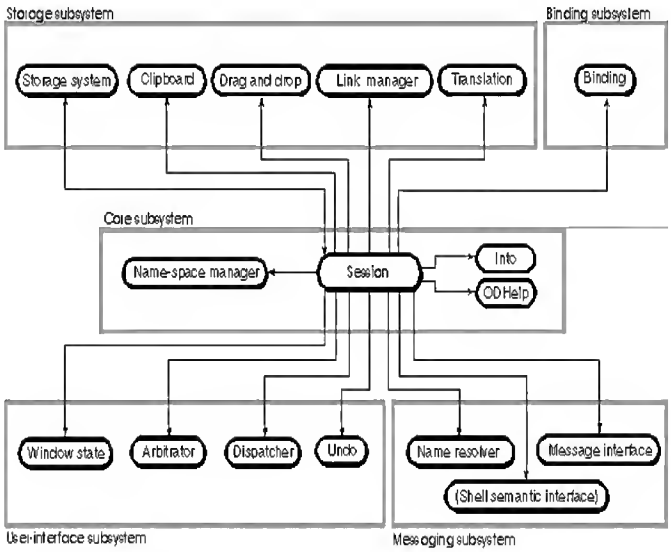
Run-Time Object Relationships

The run-time state of an OpenDoc document involves relationships among a variety of objects instantiated from the OpenDoc classes (see the figures in A Set of Classes, Not a Framework). Taken together, the diagrams in this section show the principal run-time relationships among the major OpenDoc objects for a single document. The details of the interactions among the objects is explained throughout the rest of this book.

The objects are instantiated from groups of classes in individual shared libraries called subsystems. OpenDoc is distributed as a set of subsystems. Platform developers, in providing OpenDoc on a given platform, can implement or replace individual subsystems. Part-editor developers generally need not be concerned with the characteristics of specific subsystems. For information purposes, however, subsystem boundaries are shown on the run-time object diagrams presented in this section.

Session Object

When an OpenDoc document is open, the session object occupies a central place in the relationships among the objects that constitute, support, and manipulate the document. The following figure shows the uppermost levels of that relationship.



All figures in this book that illustrate run-time object relationships use the following conventions:

- Individual objects are represented as labeled oval boxes.
- Arrows between the boxes show object references (equivalent to pointers in C++), as defined primarily by the availability of accessor functions for them in the public interface to OpenDoc. The references as shown here may not strictly mirror the actual (private) references that exist between OpenDoc objects. (Various shades or patterns used on the arrows are for visual distinction only; they do not imply different kinds of object references).
- A mutual reference between a pair of objects is represented by an arrow with a head on each end.
- A one-to-many relationship, in which one object can simultaneously reference more than one other object of a given kind, is represented by an arrow with a double head on one end.
- In diagrams that show an embedding hierarchy (such as the figure in Embedding), objects lower in the diagram are embedded more deeply than the objects above them. Objects at the same height are at the same level of embedding.

  As the preceding figure shows, the session object maintains many (mostly one-way) relationships with other objects, allowing your part to gain access to many objects through the session object. The objects are grouped into categories according to the kinds of

tasks they perform together; the following subsections discuss and expand upon the object relationships in terms of those categories. Note also that the preceding figure is incomplete; the figure in Window State and Windows and the figure in Document Storage continue the object-reference hierarchy.

Session-Level Objects

As shown in the preceding figure, the session object instantiates and maintains direct references to the following objects, accessible globally to all parts in a document.

- OpenDoc uses the binding object to pick the proper part editor for each part in a document, both when the document is opened and whenever a new part is added to it. Binding is further discussed in Binding.
- Both OpenDoc and part editors use the translation object, a wrapper for available platform-specific data translation services. If no editor is available that can manipulate the data of a part, or if the user wants the data in a different format, OpenDoc or a part editor can translate the data of a part into a part kind for which a specific translator exists. See Translation and Binding with Translation for more information.
- The Name Space Manager keeps track of all name space objects, which themselves contain tables of information needed for registration purposes. See Name Spaces for more information.
- The window state object is a list of all the open windows in which OpenDoc parts are displayed. The window state object references each open window; see the figure in Window State and Windows (The window state object also appears in figure in User-Interface Objects, in relation to others objects of the user-interface subsystem).
- The storage system object, which controls persistent storage for the session. The storage system references one or more container suites; see figure in Document Storage.
- The clipboard object is a session-wide object that represents the clipboard. It references a storage unit that represents the data held on the clipboard.
- The drag-and-drop object is also a single session-wide object. It references a storage unit that represents the data to be transferred by a drag operation.
- The dispatcher accepts user events from the underlying operating system through the document shell and dispatches them to part editors. It references dispatch modules, as shown in figure in User-Interface Objects.
- The arbitrator negotiates temporary ownership of a focus, the designation of a shared resource such as the menu bar, keystroke stream, and so on. It references focus modules, as shown in figure in User-Interface Objects.
- The undo object stores command histories for all parts in the document. It allows parts to reverse or restore the effects of multiple previous user commands. There is a single, session-wide undo object used by all parts in a document.
- The name resolver resolves object specifiers into particular objects on which semantic events can operate.
- The message interface transfers semantic events into and out of parts. If a part editor needs to send (rather than receive) semantic events, it uses the message interface. Other objects related to semantic events are shown in figure in Extension Objects and Semantic Events.
- The Info object represents the Document Properties notebook, an extensible notebook used by part editors. Other objects related to extensions are shown in figure in Extension Objects and Semantic Events.
- The Link Manager keeps track of cross-document links, facilitating the transfer of information between the source and destination parts. The Link Manager is used only by the document shell and container applications; parts never need access to it. The Link Manager is built on platform-specific facilities. Other objects related to links are shown in figure in Part Storage.
- The shell semantic interface is an instance of a subclass of ODSemanticInterface that represents the scripting support built into the document shell.

If your part needs to access any one of these session-level objects, first obtain a reference to the session object by calling the GetSession method of your part's storage unit. Then call the specific method of the session object (such as GetClipboard) that returns a reference to the needed object. To facilitate repeated access to the session object, you might store the results of GetSession in a part field with a name such as fSession.

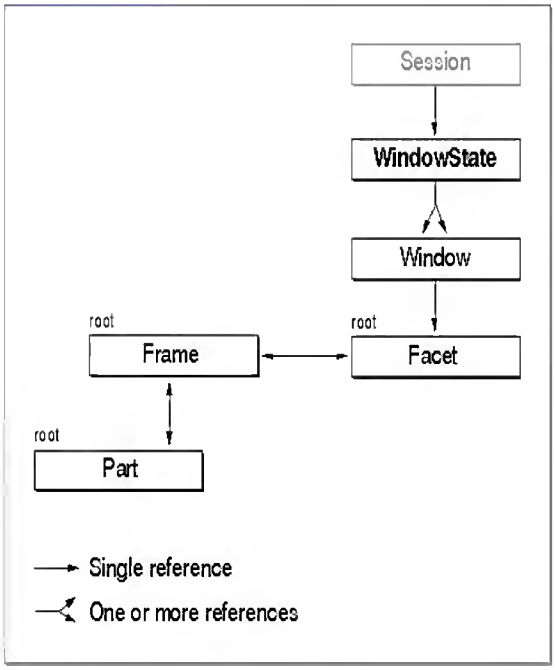
Drawing-Related Objects

The objects that make up the OpenDoc drawing capability provide a set of platform-independent protocols for embedding (placing embedded

frames within the content of a part), layout (manipulating the sizes and locations of embedded frames), and imaging (making part content and frames visible in a window). They describe part geometry in a document and provide wrapper objects for certain platform-specific imaging structures, whether for printing or for screen display.

Window State and Windows

The following figure is a simplification of the run-time object relationships among the objects involved with a document window. It is a continuation of one branch of the figure in Session Object, and shows the upper-level relationships between a window and the parts it contains.



The window state object, referenced by the session object, references one or more window objects, which are wrappers for platform-specific window structures.

Each window object references a single facet object (the root facet), the visible representation of the frame object (the root frame) of the outermost part (the root part) in the document being displayed in the window. The root frame in turn references the root part, which controls some of the basic behavior (such as printing) of the document. The root frame and root facet fill the content region of the window.

This relationship of window to root facet, root frame, and root part applies regardless of whether the window is a document window or a part window opened up from an embedded frame.

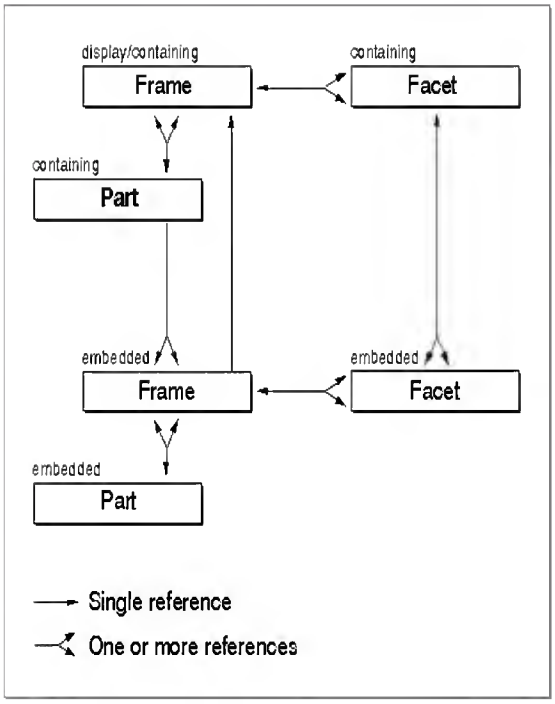
The embedding relationships among facets, frames, and parts shown the preceding figure is not complete; it continues down through all parts in the document. The root facet references its embedded facets, and the root part references its embedded frames, as shown in the next section.

Embedding

The following figure shows the relationships among facet, frame, and part objects at any level of embedding in a document. It is a direct continuation of the preceding figure but also applies at any level of embedding. In this and all other object diagrams that show embedding relationships, objects lower in the diagram are embedded more deeply than the objects above them.

The first thing to note from the following figure is that embedding is represented by two separate but basically parallel structures.

- On the left, each part references one or more display frames above it (the frames within which its contents are displayed) and one or more embedded frames below it. Those embedded frames, in turn, reference the parts for which they are the display frames. The document embedding structure, then, is represented by a frame-to-part, frame-to-part sequence in which each part only indirectly references its embedded parts. Furthermore, an embedded frame does not directly reference its containing part; instead, it references a display frame of its containing part.



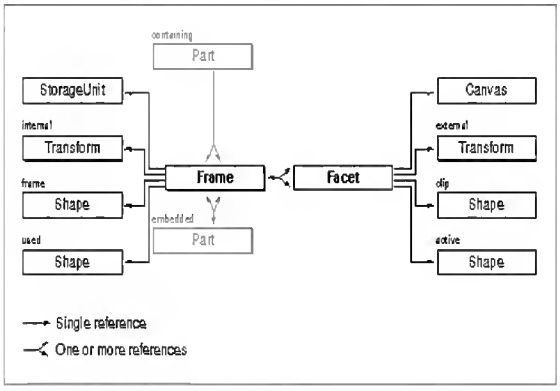
- On the right side of the following figure, the facets for the frames form their own, simpler hierarchy. Each containing facet directly references its embedded facets, and each embedded facet directly references its containing facet. This more direct imaging hierarchy allows for fast event dispatching by OpenDoc.

Connecting the two hierarchies are the frame-to-facet references. Each frame that is visible references the facet or facets that correspond to it. (A frame can have more than one facet). Each facet likewise references its frame. Note that facets need not exist unless their frames need to be drawn.

For a specific example of this embedding-object relationship in a given document, see the figure in Frame and Facet Hierarchies.

Layout and Imaging

The frame and facet objects shown in the figure in Embedding have additional references besides those shown. A part's display frame (or frames) and facet (or facets) use several other OpenDoc objects when laying themselves out and preparing to draw the content of their part. The following figure shows these additional relationships, for a given frame-facet pair at any level of embedding.



Each display frame object for the part being drawn can include references to these objects:

- A transform object (the *internal transform*) that defines the geometric offset or transformation of the part within its frame
- A shape object (the *frame shape*) that defines the basic shape of the frame

- Another shape object (the *used shape*) that defines the portion of the frame shape that is actually drawn into
- A storage unit, for storing the state of the frame into its document (unless it is a nonpersistent frame)

Each of the facet objects for the part being drawn represents an area within a window (or printer image) that corresponds to a visible display frame (or part of a frame) of the part. The facet can include references to these other objects, which hold platform-specific drawing information:

- A *canvas* object, which describes a platform-specific drawing context or structure
- A *transform* object (the *external transform*) that defines an external geometric offset or transformation for the facet within the containing part
- A *shape* object that defines the clip shape for the facet (what portion of the frame shape is drawn)
- Another shape object that defines the active shape for the facet (what portion of the frame shape accepts events)

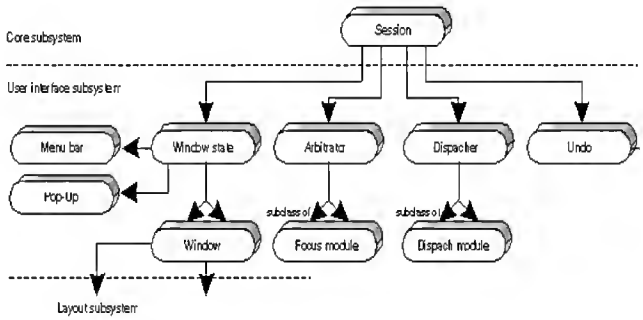
During the layout and imaging process, a part editor is typically asked to draw the contents of a particular facet. The part editor gets the clipping, transformation, and layout information from the facet and its frame, and then makes platform-specific graphics calls to perform the actual drawing.

For a more complete description of the relationships among frames, facets, and parts, see Frames and Facets. For a more complete description of the drawing process, see Drawing.

User-Interface Objects

Run-time relationships in the user-interface subsystem include some objects not directly referenced by the session object. The following figure extends a portion of the figure in Session Object to show these additional objects:

- The window state object references (in addition to all window objects) the menu bar object, which represents the base menu bar created by OpenDoc. Your part copies and adds to this menu bar to create its menus. It also references the pop-up menu object that represents the base pop-up menu created by OpenDoc. Your part copies and adds to this pop-up menu to create its pop-up menus.
- The dispatcher references one or more dispatch modules, which control which frame or part handles each event. The dispatching system is modular; you can extend it to handle new classes of events by adding dispatch module objects.
- The arbitrator references one or more focus modules, which allocate temporary ownership of shared software or hardware resources. Like the dispatcher, the arbitrator is modular; through addition of focus modules, you can extend it to handle new classes of shared resources.

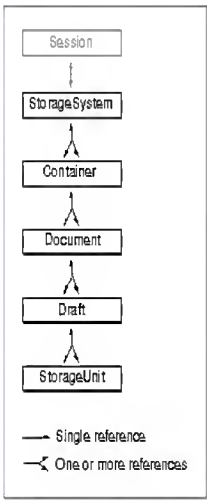


Storage Objects

The storage capabilities of OpenDoc include both document storage and data transfer. As the figure in Session Object shows, three objects directly referenced by the session object are involved with storage issues.

Document Storage

OpenDoc manages persistent storage for parts and other objects in documents. Storage in OpenDoc consists of structured storage elements that can contain many data streams. The following figure shows the main storage-related objects and their run-time relationships. It is a continuation of one branch of the figure in Session Object.



These objects combine to make up a *container suite* , a specific implementation of the OpenDoc storage architecture. Container suites can be implemented in different ways on different platforms, and need not be limited to file-based systems. Here is how the objects relate to each other:

- The *storage system* object, referenced by the session object, instantiates and maintains a list of container objects. Each container may contain one (or possibly more, depending on the capabilities of the container suite) document objects, each of which represents an OpenDoc document.
- Each document contains one or more draft objects. Each draft is unique, and represents a snapshot of the document's state at a particular moment.
- Each draft contains a number of storage unit objects. Each storage unit can contain several different data streams, all of which provide information about the object to which the storage unit applies. The data streams, or values, in a storage unit are identified by property, name (the kind of information contained) and value type, (the data type of that information). A storage unit can hold more than one property, and a property can hold streams of more than one value type.

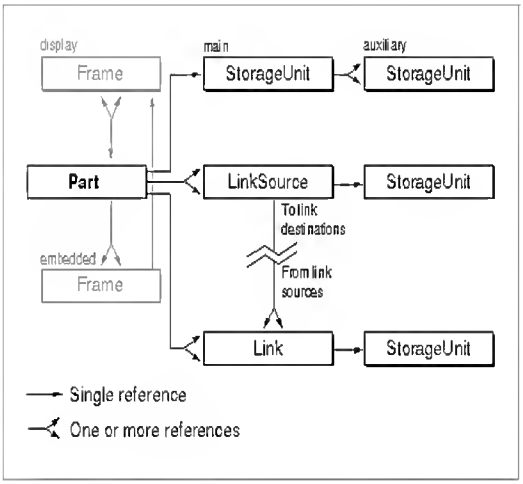
Besides being referenced by its draft, each storage unit is also referenced by the object whose data it stores. As the figure in Part Storage shows, for example, a part has a reference to its main storage unit.

Storage units and the storage system are described in more detail in Storage.

Part Storage

The following figure shows additional object references maintained by the part object, beyond those shown in the figure in Embedding that facilitate data transfer.

- A part object that contains the source of a link references a link source object, instantiated by the part that contains the source data. The link source references (a) a storage unit that holds a copy of the source data for the link and (b) one or more link objects.



- A part object that contains the destination of a link references a link object. The link object references a storage unit that holds a copy of the source data for the link. (Links and link sources for an intra-document link share the same storage unit).

The preceding figure also shows how a part must organize the storage of its content. To store its data in its draft, the part has a reference to its one main storage unit, which in turn can reference other auxiliary storage units.

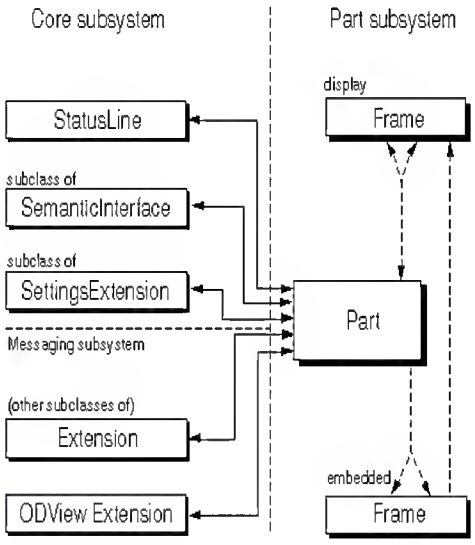
Storage for data transfer, and other data transfer issues, are described in more detail in Data Transfer.

Extension Objects and Semantic Events

OpenDoc provides a flexible method for extending its capabilities through an extension interface. Extensions to objects such as parts provide interfaces through which callers can access additional functionality.

The following figure shows additional object references that a part object can maintain, over and above those shown in the figure in Part Storage and the figure in Embedding.

- Part editors expose their extended interfaces to other parts through an extension object (a subclass of ODExtension). A part can define an extension for any purpose.



- If a part supports semantic events (scripting), it has a reference to its semantic interface object, an extension object implemented as part of OpenDoc.
- If it provides custom Properties notebooks, to allow users to change part settings, the part includes a reference to its subclass of the settings extension object. The settings extension is also a subclass of ODExtension.

The Document Shell

The document shell is the OpenDoc object that is responsible for handling document-wide and draft-wide operations. It is the closest OpenDoc run-time equivalent to a conventional application, although the same document shell handles all kinds of OpenDoc parts and documents. One of its main responsibilities is to accept events from the operating system and pass them on to OpenDoc for dispatching. Another is to handle certain menu items.

This section summarizes the run-time operations and responsibilities of the document shell and its relationship to part editors. It also describes how the document shell handles commands from the Document menu.

The discussion in this section is largely for informational purposes; your part editor never has to call the document shell.

Document Shell Operations

In OpenDoc, the document-not any application-is the owner of the process in which the document is opened. Under OpenDoc the document manages standard tasks like handling the event loop, managing files, and interacting with system menus, dialog boxes, and so on. This behavior is provided by an executable called the document shell. Individual part editors implicitly use this code in the document shell. For this reason, part editors can be significantly smaller than conventional applications. However, part editors must cooperate with each other in ways that conventional applications normally do not.

The document shell represents the process in which the parts of a single OpenDoc document execute. It is instantiated when an OpenDoc document is launched, and it is released when the document is closed. The document shell's basic responsibilities are the following:

- It creates and initializes the session object.
- It opens a document (as instructed by the user) from storage, and provides a window for it.
- It accepts user semantic events, and passes them to the OpenDoc dispatcher.
- It handles most items on the Document menu.

Opening a Document

When a document is opened by the user, OpenDoc generates a new process that instantiates the document shell. The document shell instantiates the session object, which in turn instantiates the globally available OpenDoc objects, such as the arbitrator and the window-state object. The document shell then opens the document file, and reads the document object and its most recent draft object into memory. The document shell then reads in the draft's window-state object to reconstruct the document's windows. For each window, the window state reads in the root frame.

The root frame for the window reads in the root part. The root part creates and registers the window, determines which of its embedded frames are visible, reads them in, and constructs facets for them by calling the root facet's CreateEmbeddedFacet method. The embedded frames read their own parts. Those parts in turn read in their embedded frames and create facets for them, and so on until all visible frames, parts, and facets have been read from storage.

Once all of the necessary objects are read in, OpenDoc asks each facet to draw itself; the part editor of the facet's frame draws the part content that is visible in the facet.

Saving and Reverting a Document

When the user saves a document, the draft object asks each part in the document to write itself to the document file. Each part writes its data and references to its embedded frames. The document shell then writes the window state to the document file.

When a document is saved, its data is saved in the context of the current draft. (Each draft of a document includes its own window state, allowing the user to open earlier drafts for viewing at any time).

If the user specifies that a document is to be saved as stationery, the document shell sets a flag in the document file, to notify the Operating System that a document is stationery instead of an ordinary OpenDoc document.

Reverting a draft means throwing away any changes that have been made since the last save. The document shell releases the existing window state and restores the window state from the previously saved draft. As a result of reverting a draft, the active part and thus the selection and user-interface elements may change.

Closing a Document

The document shell closes a document when its last document window is closed. As a result of this closing, some other OpenDoc document window may become active, with its own active frame and selection.

If changes are to be saved when the document closes, the document shell follows the procedures described in the previous section. If the user closes a document without saving changes to the current draft, the document shell releases the window state, leaving the document in the state it occupied at the last save.

Handling User Events

The document shell receives all user events directly and is responsible for seeing that events intended for individual parts get to the proper part editor.

The document shell passes all events to the dispatcher for dispatching to the appropriate part, without first classifying them. The dispatcher rejects events it can't identify, returning them to the shell to handle.

To dispatch an event, the dispatcher locates the appropriate dispatch module for the event and asks it to dispatch the event. The dispatch module actually distributes the event to the appropriate part or document, in cooperation with the window state and the arbitrator, using the facet hierarchy. (See User Events for more information).

Part of the document shell's event-handling job is to provide information to part editors about mouse tracking. The document shell also supplies the basic menu bar and pop-up menu used by OpenDoc documents and parts.

The document shell can handle standard window events, such as a click in the close box, although the dispatcher gives the root part of the window a chance to handle such events first. See Handling Window Events.

Menu events go first to the document shell; the shell handles most Document-menu commands, and passes others (see Document Menu) to the dispatcher for sending to the appropriate part.

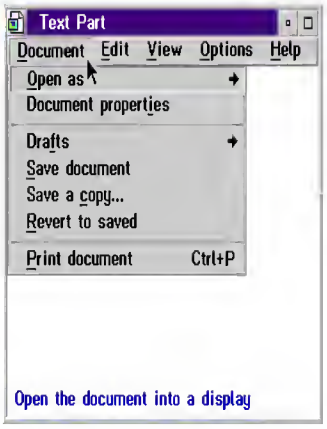
Handling Semantic Events

Semantic events sent to a part in a document are received first by the document shell and then passed to the dispatcher to be distributed in the same manner as user events. See Writing Semantic-Event Handlers.

The document shell has its own semantic interface, through which it handles the requires set of OSA events. See the note **Document shell semantic interface** in OpenDoc Semantic Interface.

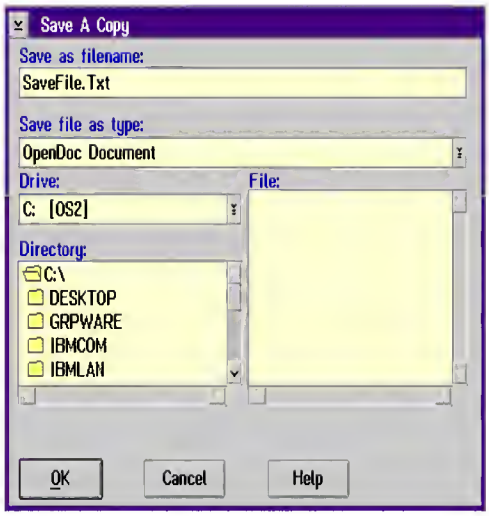
The Document Shell and the Document Menu

The following figure shows the standard OpenDoc Document menu. The OpenDoc document shell creates the standard document menu and handles most items in it. Items not handled by the document shell are handled by the root part or the active part, as described in Document Menu.

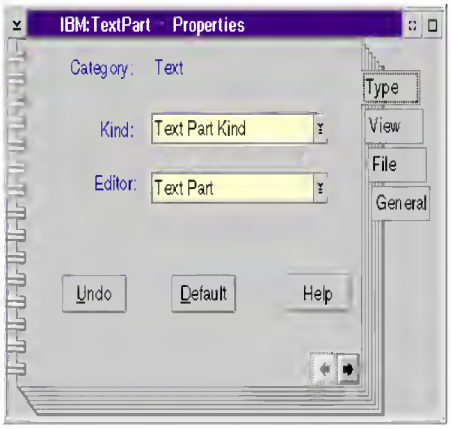


Here is how the document shell handles each item in the menu:

- **Save document.** The document shell saves the current document (the document containing the currently active part). If this is the first save of the document, the document shell displays a standard-file dialog box to allow the user to choose a document name and location.
- **Save a copy.** The document shell saves a copy of the current draft into a new file. The document shell displays a File dialog box (the following figure) to allow the user to specify the name and choose a location. The current document remains open.



- **Revert to saved.** The document shell reopens the last saved version of the current document, replacing the presently open version of it.
- **Draft history.** The document shell displays the draft history of the current document in a Details view window. When the user chooses the **Create draft** command, the document shell displays a dialog to create a new draft. An example of the Drafts History window is shown in the figure in Document Drafts.
- **Document properties.** The document shell brings up the Properties notebook of the root part, shown in the following figure.



The user can then perform several actions, including changing the view type, part kind, and editor for the root part.

- **Print document.** (Handled by the root part; see Document Menu).

Embedding Checklist

This appendix summarizes the capabilities that your part editor must have if it is to support embedding. If you want your parts to be container parts, you must implement these features in your code.

Keep in mind that handling embedded parts usually means handling embedded *frame*s ; in most cases, your part interacts only with the frames of the parts embedded in it. Those embedded frames in turn interact with the parts displayed in them, and the embedded parts interact with their own embedded frames, and so on down through the embedding hierarchy.

This checklist is meant for quick reference only. The items in the list are discussed in greater detail elsewhere in this book.

Content Model and Storage

- Add embedding to your part's content model. Create a content element that you can track and manipulate to represent an embedded frame. You might create a proxy object that can represent a frame in both its stored and in-memory state.
- Maintain a list of the frames embedded in your part. Implement an iterator (a subclass of ODEmbeddedFramesIterator) to allow callers access to the frame objects in the list. Implement your part's CreateEmbeddedFramesIterator method, through which callers instantiate the class.
- When you write your part to storage, write persistent references to your embedded frames into your part's content stream. (Do not write them into a separate property.) When you read your part from storage, reconstruct your embedded frames from the stored persistent references in your content.

  Note: For efficiency, you can use lazy instantiation for this purpose. Create a frame object for each embedded frame only when you need to display it. Purge unneeded frames when requested.

Embedded-Frame Management

- You control the sizes and locations of your embedded frames. Support frame negotiation, if appropriate, in your RequestFrameShape method.
- Be prepared to receive requests for additional frames from an embedded part. Implement your RequestEmbeddedFrame method, and handle calls to it as appropriate for your content model.

  If you add a requested frame, use the requested shape for both sizing and relative positioning of the frame, if your content model allows it. Normalize the frame shape that you return. Give the frame an appropriate group ID and sequence number.

  If you grant requests for additional embedded frames, implement your part's RemoveEmbeddedFrame method so that embedded parts can have you delete those frames when they are no longer needed.
- If you create an embedded frame that displays the same content in the same presentation as another embedded frame, use AttachSourceFrame to have the embedded part attach the new frame to the original (source) frame, so that updates in one will be reflected in the other.
- When you no longer need an embedded frame, permanently remove it by removing all of its facets, setting its containing frame to null, and calling Release or Remove method, as appropriate.
- Another part may request that you scroll you part's content to reveal an embedded frame. You may have to accomplish this by scrolling your own content, or by requesting that your containing part scroll its content. Implement your part's RevealFrame method to fulfill this request.

- Support undo of all embedded-frame manipulations.

Depending on your part's content model and features, you might also want to add support for these capabilities:

- If you support sequenced frames, use frame groups to define sets of sequentially ordered embedded frames. Assign sequence numbers so that the embedded part can know in what order to fill its frames with content.
- If your content model suggests that you should make any of your content properties (such as, for example, text font or style) available for adoption by embedded parts, implement your AcquireContainingPartProperties method so that embedded parts can call it. Likewise, call your embedded parts' ContainingPartPropertiesUpdated method whenever you change any of your adoptable content properties.
- Whether or not you support linking, implement your part's LinkStatusChanged method, so that you can call your embedded frames' ChangeLinkStatus methods and communicate changes in your own display frame's link status.

Drawing

- Define the view type (icon or frame) that embedded parts should have. By convention, most embedded parts should have frame view type.
- Create facets for all of your visible embedded frames. Add facets as frames become visible, remove facets or mark them as purgeable when frames are no longer visible.
- When a frame embedded in your part is the active frame, you need to clip whatever content the active frame border obscures, and you also need to clip the active frame border where your content obscures it. Implement your part's AdjustBorderShape method for this purpose.
- Support selection of your embedded frames. Highlight a selected embedded frame by drawing the selected frame border (and resize handles, if you support resizing of embedded frames) on the embedded frame's frame shape, not its used shape.
- When a range selection includes an embedded frame, give the embedded part's facet the highlight style the embedded part should adopt to be consistent with your highlighting model.
- When you part is displayed with a thumbnail view type, you are responsible for integrating miniature representations of any visible embedded parts into the thumbnail.

Depending on your part's content model and features, you may also want to add support for these capabilities:

- If you support overlapping of embedded frames and intrinsic content, maintain the proper clip shapes for all of your embedded frames' facets. Maintain a z-ordered list of your embedded facets, and use it to account for overlaps with other embedded frames and with your part's content elements.
- If you wrap your content to the used shape of your embedded parts, implement your part's UsedShapeChanged method, so that embedded parts can notify you when that used shape changes.
- If you create offscreen canvases, implement your part's CanvasUpdated method so that you can transfer your embedded parts' asynchronous drawing to you parent canvas.
- If you support split-frame views of embedded parts, provide multiple facets for your embedded frames.

Event Handling

- Handle mouse events that occur within and on the borders of an embedded frame as needed. Use these events to initiate a drag of the embedded frame, to select the embedded frame, or to activate your own part.
- If you want to receive events sent to an embedded frame that its part chooses not to handle, set the doesPropagateEvents flag of the frame when you create it.
- Use mouse-down events in resize handles of selected embedded frames to initiate resize operations.
- Enable and respond to these items in the Edit menu:

- **Undo**, **Redo**: include embedded frames in the actions that you save and restore. Handle any necessary reading and writing as described under data transfer (below).
- **Cut**, **Copy**, **Paste**, **Paste as**: include embedded frames in the data you manipulate. Handle as described under data transfer (below).
- **Selection properties**: Display a selected embedded frame's Part Properties notebook.
- **Open selection**: Open selected embedded parts into a window.
- **Show selection as**: Set view type of selected embedded frames.

Depending on your part's content model and features, you may also want to add support for these capabilities:

- If you want to receive events sent to an embedded frame that its part chooses not to handle, set the event-propagating flag of the frame when you create it.
- If you support resizing of embedded frames, use mouse events in resize handles of selected embedded frames to initiate resize operations.

Data Transfer

General

- Modify your basic routines for writing and reading data to account for embedded parts. This includes transfers involving the clipboard, the drag-and-drop object, link-source objects, link objects, other parts (as per the **Insert** command), and fulfilled promises.
  - Do all reading and writing in the context of cloning. In addition to reading and writing intrinsic data, clone embedded frames (and other referenced objects) to or from the storage units involved.
  - If you write a single embedded part, you should also write its frame as an annotation to the storage unit. If you read a single embedded part, use that annotated frame as the frame of the part you read.
  - Modify your part's CloneInto method to add cloning of your part's embedded parts.
- Follow the conventions for receiving transferred data. Incorporate the data if your part editor can read it; otherwise, embed it as a separate part.
- Modify your handling of the Paste As dialog box: allow the user to select the Embed as button.

Cutting Data

- If you cut (or drag-move) data from your part that includes embedded frames, keep these cautions in mind:
  - Do not assume that removing one embedded frame removes a part entirely; other frames of the part may still be embedded in your part.
  - Be sure to set the containing frame of each removed frame to null, since it is no longer embedded in your part.
  - Use your own display facet's embedded facet iterator to delete the facets of each frame that was removed.
  - Be sure to remove, rather than release or delete, each cut frame object. To support undo, do that only when your part's DisposeActionState method is called.

Linking

- Implement your part's EmbeddedFrameUpdated method, so that embedded frames can notify your part when their content has changed and your part can in turn update any link sources involving that embedded content.
- Implement your part's EditInLinkAttempted method, so that embedded parts can notify you when the user attempts to edit their data when it is part of a link destination that you maintain.
- Call the ChangeLinkStatus method of any embedded frame that becomes part of, or ceases to be part of, a link source or destination that you maintain.

For Scripting Support

- In your content model, create a content object that represents an embedded frame. Create an object accessor for embedded frames, if you want access to any custom information about your embedded frames. (OpenDoc provides a default accessor that can access an embedded frame or any of its standard user properties.)
- Write a semantic-event handler that can manipulate any of an embedded frame's properties that you define, such as its position or its selection state. (Semantic events involving standard user properties of the embedded frame's part will be passed to the part itself.)
- Implement your part's EmbeddedFrameSpec method, to create an object accessor for your display frame.

Part Developer's Checklist

This checklist contains questions about the OpenDoc user interface that you can ask yourself while reviewing your software. These questions will help bring to mind the particulars of the guidelines.

You must be able to answer every question "yes" to ensure conformity with the guidelines. However, to provide the most usable interface, sometimes you need to make tradeoffs in your application. Remember to maintain the spirit of the guidelines when reviewing your product.

Splash Screens

- Do you display a splash screen more than once a day, either upon software installation or first software use of the session?
- Is the splash screen unobtrusive, and does it disappear automatically (without user interaction)?

Menus

- Do you enable/disable menu items on the menu bar to reflect the current state of your part?
- Do you add/remove menu items on the popup menu to reflect the current state of your part?
- Do part-supplied menu items provide no more than two levels of cascaded menus?
- Do you adjust your menus for single/multiple selection?
- Do you display prompt line help for part-supplied menu items?

Document Menu

- Have you maintained the standard Document menu except to add items that apply to the active document?
- When your part is opened into a part window, do you remove the Document menu and replace it with your part's menu?

  If you have no additional menus for part windows, have you disabled the Document menu when your part is opened into a part

window?

- If you are a container, do you adjust the Document menu of the current menu bar when your part is the root part, but not active?

Edit Menu

- Have you maintained the standard Edit menu except to add items that involve changing part content?
- Does the **Undo** command reverse the effects of the last undoable user action and restore all parts to their states before that action?
- Does the **Redo** command reverse the effects of the last undo action and restore all parts to their states before that undo?
- Does the **Cut** command remove the selection and place it on the Clipboard?
- Does the **Copy** command copy the selection to the Clipboard?
- Does the **Paste** command place the contents of the Clipboard at the insertion point?
- Does the **Paste as** command display the Paste As dialog?
- Does the **Paste link** command insert the link from the clipboard?
- Does the **Delete** command remove the selection?
- Does the **Select all** command select all the contents of the active part?
- Does the **Deselect all** command remove all selections from within the active part?
- Does the **Break link** command remove all links from the active part?
- Does the **Create** command create a new instance of your part and place it on the clipboard?
- Does the **Open selection** command open the selected component or components into a window?
- Does the **Selection properties** command open a properties notebook for the selected component or components?
- Does the **Show selection as** command change the view type of the selected component or components?

View Menu

- Does the **Show frame outline** command appear in the View menu only when a frame is opened into a part window (via Open As ->Window) and your part content is larger than is displayed in its frame? Does the **Show frame outline** command display an outline of the frame in the window to show the portion of the part's contents that is visible in the frame? Can the user drag this outline to adjust the visible region of the part within the frame?
- Does the **Hide frame outline** command appear in the View menu instead of the **Show frame outline** after a user has chosen **Show Frame Outline**? Does the **Hide frame outline** command remove the display of the outline of the frame in the part window?
- Is the **Show links** menu item enabled when there are links within your part? Does the **Show links** command display link borders when chosen by the user?
- Does the **Hide links** command appear in the View menu instead of **Show links** after a user has chosen **Show Links**? Does the **Hide links** remove the link borders?
- If you are a containing part, do you support the **Open as icons**, **Open as tree**, and **Open as details** views?

Help Menu

- Do you place your part information dialog at the end of the Help menu, after the product information dialog?
- If you supply a part tutorial, have you placed the tutorial menu item above the product and part information dialogs?

Icons

- Do all document, template, part editor, and part view icons conform to the standard sizes?
- Do you provide the following part icons:
    - Large icon
    - Small icon
- Do thumbnail icons accurately represent the part's contents?

Part Properties

- Does the properties notebook show that your part correctly supports the following properties?
    - Category, stating the type of data in a part
    - Kind, the data format of the part's contents
    - Editor, a pointer to the editor being used to edit the part's contents
    - Name, a text string identifying the part
    - Icon, the icon that represents this part

Part Selection

- Does an active part become selected after a user clicks on its border?
- Does a bundled, unselected part become selected after a user clicks anywhere within its frame?
- Does your active part display the specified selected highlight of its selected embedded parts?
- If a user selects a part inside an active container using Alt+Mouse Button 1, does that part become selected? If there is a previous selection, does the active part deselect the previous selection?

Resizing Frames

- Can users resize a frame by selecting it and dragging any selection handle on the frame's border?
- Do you provide cursor feedback when the user resizes a frame?

Changing Display Forms

- When the user selects a part icon and chooses **Show selection as** frame from the Edit menu, does the part appear as a frame?
- When the user selects a frame and chooses one of the icon or thumbnail representations from the Show Selected As menu, does the part's frame change into an icon?
- When the user activates a frame and chooses one of the icon or thumbnail representations from the View Show As menu, does the part's frame change into an icon?
- If your part receives a message about the preferred display form of its containing part, do you draw your part accordingly?
- When the user changes the display form of your part from the properties notebook and then closes the notebook, does your part change its display form to that indicated by the user?

Multiple Selection

- Can users select multiple embedded parts within your active part by holding the Shift or Ctrl keys and clicking on multiple icons or frames? Does a click on an already selected part remove the part from the selection?
- If your part supports dragging to select, can users select multiple parts within an active part by dragging across the parts? If the user drags across parts while holding the Shift or Ctrl keys, are previously selected parts removed from the selection and unselected parts added to the selection?

Drag and Drop

- Can users drag a selected item after placing the cursor anywhere within the item?
- When a user selects an item and drags it to a new location within a document, does the item appear in the new location?
- When a user selects an item and drags it to another document, is the item copied into the document?
- When a user selects an item and drags it to the desktop, does a copy of the item, displayed as an icon, appear on the desktop?
- Does holding down the Ctrl key during a drag operation force a move operation?
- If the user holds down the Ctrl + Shift keys while dragging part content or a part, do you display the Paste As dialog, enabling creation of a link?
- Do you correctly incorporate or embed content as a result of a drag and drop operation?

Scrolling

- When a user starts to make a selection in an embedded part partly obscured by the window border and extends the selection by moving the pointer outside the window in the direction of the obscured portion, does the containing part's area scroll to reveal the content? Does scrolling stop when the user moves the pointer back inside the frame or when the border of the embedded part's frame is visible?
- If you support scroll bars for your part, can users hide them to see the document as it will print?

- Does clicking in a scroll bar in your active part cause its content to scroll immediately?

Sequenced Frames

- If your part supports sequenced frames, does all of the content in the frames appear in a part window when the user opens one of the sequenced frames? Do you provide scroll bars in the part window?

Linking

- When the user chooses the **Paste as** command, do you allow a link destination to be created in your part?
- When a user selects the **Show links** command, does your part display the specified link borders?
- When the user selects the **Hide links** menu item, do you remove the link borders?
- When the user selects content that contains a link, do you display the link border? If the user clicks outside of the linked content, do you remove the link border?
- When the user clicks on the border of a link, do you select the link and removed any content selection?
- Can users use the Cut and Paste menu items to move a link?
- Can users use the Copy and Paste menu items to copy a link?
- If the user tries to change the content of a link destination that will be destroyed when the destination receives an update from the link source, do you display a warning to the user?
- When the user selects **Break link** from the menu, do you remove the link?
- Do you delete links after a user selects the source and/or destination of a link and chooses the Cut menu item or presses the Delete key?
- Do you update an automatic link in the same document when the source content changes?
- Do you update a manual link when the user chooses Update Now from the Properties notebook when it is the link source?
- Do you update a manual link when the user chooses Update Now from the Properties notebook when it is the link destination?
- Do you delete links after a user selects the source or the destination of a link and chooses the cut menu item or presses the Delete key?
